1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	5 Dec 89 to 5 Dec 90	Final

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Tartan Laboratories Incorporated, Tartan Ada VMS/C30, Version 2.2, VAX Station 3100 (host) to 320c30 on TI Application Board (target) 891205T1.10260	5. FUNDING NUMBERS
6. AUTHOR(S) IABG-AVF Ottobrunn, FEDERAL REPUBLIC OF GERMANY	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrieanlagen-Betriebsgeselschaft Dept. SZT Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY	8. PERFORMING ORGANIZATION REPORT NUMBER AVF-IABG-058
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Tartan Laboratories Incorporated,Tartan Ada VMS/C30 Version 2.2,Ottobrunn West Germany, VAX Station 3100 under VAX/VMS 5.1 B (Host & Target), ACVC1.10

DTIC
ELECTE
JUN 27, 1990
S   B   D

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office		15. NUMBER OF PAGES
		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #891205I1.10260
✓TARTAN LABORATORIES INCORPORATED/
Tartan Ada VMS/C30, Version 2.2,
VAX Station 3100 to 320C30 on TI Application Board

Completion of On-Site Testing:
5 December 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

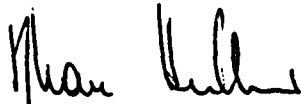Compiler Name: Tartan Ada VMS/C30 Version 2.2

Certificate Number: #891205I1.10260
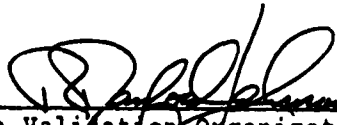
Host: VAX Station 3100 under VAX/VMS 5.1B

Target: 320C30 on TI Application Board

Testing completed 5 December 1989 using ACVC 1.10.

This report has been reviewed and is approved.

_____

Dr. S. Heilbrunner
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

_____

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA   22311

_____

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC   20301

Accession For		
NTIS GRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A-1		

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability. (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by IABG mbH, Abt SZT according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO).

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> IABG mbH, Abt SZT
> Einsteinstr 20
> D8012 Ottobrunn

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language,
    ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint
    Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech,
    Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada
              programs that tests the conformity of an Ada compiler to the
              Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary   point addressed by a comment on the Ada Standard.   These
              comments are given a unique identification number having the
              form AI-ddddd.

Ada Standard ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility.   The AVF is responsible for
              conducting   compiler validations according to procedures
              contained in the Ada Compiler Validation Procedures and
              Guidelines.

AVO   The Ada Validation Organization.   The AVO has oversight
              authority   over   all AVF practices for the purpose of
              maintaining a uniform process for validation of Ada
              compilers. The AVO provides administrative and technical
              support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this
              report, a compiler is any language processor, including
              cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that
              demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

3

Inapplicable   An ACVC test that uses features of the language that a
test   compiler is not required to support or may legitimately
                support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected
                result.

Target   The computer which executes the code generated by the
                compiler.

Test   A program that checks a compiler's conformity regarding a
                particular feature or a combination of features to the Ada
                Standard. In the context of this report, the term is used to
                designate a single test, which may comprise one or more
                files.

Withdrawn   An ACVC test found to be incorrect and not used to check
test   conformity to the Ada Standard. A test may be incorrect
                because it has an invalid test objective, fails to meet its
                test objective, or contains illegal or erroneous use of the
                language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC
contains both legal and illegal Ada programs structured into six test
classes: A, B, C, D, E, and L. The first letter of a test name identifies
the class to which it belongs. Class A, C, D, and E tests are executable,
and special program units are used to report their results during
execution. Class B tests are expected to produce compilation errors.
Class L tests are expected to produce errors because of the way in which a
program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada
programs with certain language constructs which cannot be verified at run
time. There are no explicit program components in a Class A test to check
semantics. For example, a Class A test checks that reserved words of
another language (other than those already reserved in the Ada language)
are not treated as reserved words by an Ada compiler. A Class A test is
passed if no errors are detected at compile time and the program executes
to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage.
Class B tests are not executable. Each test in this class is compiled and
the resulting compilation listing is examined to verify that every syntax
or semantic error in the test is detected. A Class B test is passed if
every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be

5

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

    Compiler: Tartan Ada VMS/C30 Version 2.2

    ACVC Version: 1.10

    Certificate Number: #891205I1.10260

    Host Computer:

            Machine: VAX Station 3100

            Operating System: VAX/VMS 5.1B

            Memory Size: 8 MB

    Target Computer:

            Machine: 320C30 on TI Application Board

            Operating System: bare machine

            Memory Size: 512K x 32 bit words

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 10 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined type LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round towards positive infinity . (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round towards positive infinity. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round towards positive infinity. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR for one dimensional array types and two dimensional array types , and no exception for one dimensional array subtypes and two dimensional array subtypes. (See test C36003A.)

2) NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

9

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEG. ('LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This compiler enforces the following two rules concerning declarations and proper bodies which are individual compilation units:

o generic bodies must be compiled and completed before their instantiation.

o recompilation of a generic body or any of its transitive subunits makes all units obsolete which instantiate that generic body.

These rules are enforced whether the compilation units are in separate compilation files or not. AI408 and AI506 allow this behaviour.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

11

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 619 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation, and for 238 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 81 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1128	1719	16	16	46	3054
Inapplicable	0	10	596	1	12	0	619
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	192	547	485	242	171	99	159	332	133	36	252	330	76	3054
N/A	20	102	195	6	1	0	7	0	4	0	0	39	245	619
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10
at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features
that a compiler is not required by the Ada Standard to support. Others may
depend on the result of another test that is either inapplicable or
withdrawn. The applicability of a test to an implementation is considered
each time a validation is attempted. A test that is inapplicable for one
validation attempt is not necessarily inapplicable for a subsequent
attempt. For this validation attempt, 619 tests were inapplicable for the
reasons indicated:

    a. The following 285 tests are not applicable because they have
       floating-point type declarations requiring more digits than
       SYSTEM.MAX_DIGITS:

           C24113F..Y (20 tests)   C35705F..Y (20 tests)
           C35706F..Y (20 tests)   C35707F..Y (20 tests)

14

```
        C35708F..Y (20 tests)    C35802F..Z (21 tests)
        C45241F..Y (20 tests)    C45321F..Y (20 tests)
        C45421F..Y (20 tests)    C45521F..Z (21 tests)
        C45524F..Z (21 tests)    C45621F..Z (21 tests)
        C45641F..Y (20 tests)    C46012F..Z (21 tests)
```

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

```
        C45231B     C45304B     C45502B     C45503B     C45504B
        C45504E     C45611B     C45613B     C45614B     C45631B
        C45632B     B52004E     C55B07B     B55B09D     B86001V
        CD7101E
```

d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

```
        C45231C     C45304C     C45502C     C45503C     C45504C
        C45504F     C45611C     C45613C     C45614C     C45631C
        C45632C     B52004D     C55B07A     B55B09C     B86001W
        CD7101F
```

e. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 32.

f. D64005G is not applicable because this implementation does not support nesting 17 levels of recursive procedure calls.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001X, C45231D, and CD7101G are not applicable because this implementation supports no predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

k. CA2009A, CA2009C, CA2009F and CA2009D are not applicable because this compiler creates dependencies between generic bodies, and units that instantiate them (see section 2.2i for rules and restrictions concerning generics).

15

l. CD1009C, CD2A41A..E (5 tests), and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

m. CD2A61E, CD2A61G, and CD2A61I are not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

o. CD2A91A..E (5 tests) are not applicable because 'SIZE length clauses for task types are not supported.

p. CD2B11G is not applicable because 'STORAGE_SIZE representation clauses are not supported for access types where the designated type is a task type.

q. CD2B15B is not applicable because a collection size larger than the size specified was allocated.

r. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..B (2 tests)	CE2108C..H (6 tests)
CE2109A..C (3 tests)	CE2110A..D (4 tests)
CE2111A..I (9 tests)	CE2115A..B (2 tests)
CE2201A..C (3 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A

16

CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A	CE3411C
CE3412A	EE3412C
CE3413A	CE3413C
CE3602A..D (4 tests)	CE3603A
CE3604A..B (2 tests)	CE3605A..E (5 tests)
CE3606A..B (2 tests)	CE3704A..F (6 tests)
CE3704M..O (3 tests)	CE3706D
CE3706F..G (2 tests)	CE3804A..P (16 tests)
CE3805A..B (2 tests)	CE3806A..B (2 tests)
CE3806D..E (2 tests)	CE3806G..H (2 tests)
CE3905A..C (3 tests)	CE3905L
CE3906A..C (3 tests)	CE3906E..F (2 tests)

These tests were not processed because their inapplicability can be deduced from the result of other tests.

s. Tests CE2103A..B (2 tests) and CE3107A raise USE_ERROR although NAME_ERROR is expected. These tests report FAILED but they were graded not applicable because this implementation does not support permanent files.

t. EE2201D, EE2201E, EE2401D, EE2401G are inapplicable because sequential, text, and direct access files are not supported.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 79 tests.

a. The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24007A	B24009A	B25002B	B32201A	B34005N
B34005T	B34007H	B35701A	B36171A	B36201A	B37101A
B37102A	B37201A	B37202A	B37203A	B37302A	B38003A
B38003B	B38008A	B38008B	B38009A	B38009B	B38103A
B38103B	B38103C	B38103D	B38103E	B43202C	B44002A
B48002A	B48002B	B48002D	B48002E	B48002G	B48003E
B49003A	B49005A	B49006A	B49007A	B49009A	B4A010C
B54A20A	B54A25A	B58002A	B58002B	B59001A	B59001C
B59001I	B62006C	B67001A	B67001B	B67001C	B67001D
B74103E	B74104A	B85007C	B91005A	B95003A	B95007B
B95031A	B95074E	BC1002A	BC1109A	BC1109C	BC1206A
BC2001E	BC3005B	BC3009C	BD5005B		

b. For the two tests BC3204C and BC3205D, the compilation order was changed to

        BC3204C0, C1, C2, C3M, C4, C5, C6, C3M
and
        BC3205D0, D2, D1M

respectively. This change was necessary because of the compiler's rules for separately compiled generic units (see section 2.2i for rules and restrictions concerning generics). When processed in this order the expected error messages were produced for BC3204C3M and BC3205D1M.

c. The two tests BC3204D and BC3205C consist of several compilation units each. The compilation units for the main procedures are near the beginning of the files. When processing these files unchanged, a link error is reported instead of the expected compiled generic units. Therefore, the compilation files were modified by appending copies of the main procedures to the end of

these files. When processed, the expected error messages were generated by the compiler.

d. Tests C39005A, CD7004C, CD7005E and CD7006E wrongly presume an order of elaboration of the library unit bodies. These tests were modified to include a PRAGMA ELABORATE (REPORT);

e. Test E28002B checks that predefined or unrecognized pragmas may have arguments involving overloaded identifiers without enough contextual information to resolve the overloading. It also checks the correct processing of pragma LIST. For this implementation, pragma LIST is only recognised if the compilation file is compiled without errors or warnings. Hence, the test was modified to demonstrate the correct processing of pragma LIST.

For this implementation, the required support package specification, SPPRT13SP, was rewritten to provide constant values for the function names.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Tartan Ada Sun/C30 Version 2.2 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests. The applicant certified that testing results for the computing system of this validation would be identical to the ones submitted for review prior to validation.

3.7.2 Test Method

Testing of the Tartan Ada VMS/C30 Version 2.2 compiler using ACVC Version 1.10 was conducted by IABG on the premises of TARTAN. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host Computer:
        Machine: VAX Station 3100
        Operating System: VAX/VMS 5.1B
        Memory Size: 8 MB

Target Computer:
        Machine: 320C30 on TI Application Board
        Operating System: bare machine
        Memory Size: 512K x 32 bit words

Compiler:
Tartan Ada VMS/C30 Version 2.2

The original ACVC was customized prior to the validation visit in order to remove all withdrawn tests, inapplicable I/O tests and tests requiring unsupported floating point precisions. Tests that make use of implementation specific values were also customized. Tests requiring modifications during the prevalidation testing were modified accordingly.

A tape containing the customized ACVC was read by the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked. All executable tests were transferred via an RS232 line to the target computer where they were run. Results were transferred to the host computer in the same way. The results were then transferred via Ethernet to a SUN 3/60, where they were evaluated. Archiving was done on the host computer.

The compiler was tested using command scripts provided by TARTAN LABORATORIES INCORPORATED and reviewed by the validation team. The compiler was tested using no option settings. All chapter B tests were compiled with the listing option on (i.e. /LIST). The linker was called with the command

        ALBC30 LINK <testname>

The compiler was called with the command

        ADAC30 <testname>

3.7.3 Test Site

Testing was conducted at TARTAN LABORATORIES INCORPORATED, Pittsburgh and was completed on 5 December 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

TARTAN LABORATORIES INCORPORATED has submitted the following Declaration of Conformance concerning the Tartan Ada VMS/C30 Version 2.2 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Tartan Laboratories Incorporated
Ada Validation Facility: IABG mbH, Dept. SZT
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: Tartan Ada VMS/C30
Base Compiler Version: Version 2.2
Host Computer: VAX Station 3100 under VAX/VMS V5.1B
Target Computer: 320C30 on TI Application Board (bare target)

Implementor's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, have
implemented no deliberate extensions to the Ada Language Standard
ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I
declare that Tartan Laboratories Incorporated is the owner of record of
the Ada Language compiler(s) listed above and, as such, is responsible
for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A.
All certificates and registrations for Ada Language compiler(s) listed
in this declaration shall be made only in the owner's corporate name.

_____   Date:___7 Dec 89_____
Tartan Laboratories Incorporated
D. L. Evans, President

Owner's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, take
full responsibility for implementation and maintenance of the Ada com-
piler(s) listed above, and agree to the public disclosure of the final
Validation Summary Report.   I declare that all of the Ada Language
compilers listed, and their host/target performance, are in compliance
with the Ada Language Standard ANSI/MIL-STD-1815A.

_____   Date:___7 Dec 89_____
Tartan Laboratories Incorporated
D. L. Evans, President

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies corres-
pond to implementation-dependent pragmas, to certain machine-
dependent conventions as mentioned in chapter 13 of the Ada Stan-
dard, and to certain allowed restrictions on representation
clauses. The implementation-dependent characteristics of the
Tartan Ada VMS/C30 Version 2.2 compiler, as described in this
Appendix, are provided by TARTAN LABORATORIES INCORPORATED.
Unless specifically noted otherwise, references in this appendix
are to compiler documentation and not to this report.
Implementation-specific portions of the package STANDARD, which
are not a part of Appendix F, are:

    package STANDARD is

        ...

            type INTEGER is range -2_147_483_648 .. 2_147_483_647;

            type FLOAT is digits 6 range -16#0.1000_00#E+33 ..
            16#0.FFFF_FF#E+32;

            type LONG_FLOAT is digits 9 range
            -16#0.1000_0000_0#E+33 .. 16#0.FFFF_FFFF_0#E+32;

            type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

        ...

    end STANDARD;

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute. Inc., February 17, 1983) .

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.

- Pragma ELABORATE is supported.

- Pragma INLINE is supported.

- Pragma INTERFACE is now supported. The language_name is ignored by the compiler; that is, the standard Ada calling convention is used for every call to a subprogram to which this pragma is applied.

- Pragma LIST is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma MEMORY_SIZE is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).

- Pragma PACK is fully supported.

- Pragma PAGE is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma PRIORITY is supported.

- Pragma STORAGE_UNIT is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

- SHARED is not supported. No warning is issued if it is supplied.

- Pragma SUPPRESS is supported.

- Pragma SYSTEM_NAME is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (*Ada-simple-name, string-constant*)

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

This pragma has no effect when applied to a library subprogram or to a *renames* declaration; in the latter case, no warning message is given.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags simply by appending the suffix #GOTO. Therefore, the external linkage name has 5 fewer significant characters than the lower limit of other tools that need to process the name (e.g., 40 in the case of the Tartan Linker).

5.1.2.2. Pragma FOREIGN_BODY

In addition to Pragma INTERFACE, Tartan Ada supplies Pragma FOREIGN_BODY as a way to access subprograms in other languages.

Unlike Pragma INTERFACE, Pragma FOREIGN_BODY allows access to objects and exceptions (in addition to subprograms) to and from other languages.

Some restrictions on Pragma FOREIGN_BODY that are not applicable to Pragma INTERFACE are:

- Pragma FOREIGN_BODY must appear in a top-level non-generic package specification.
- All objects, exceptions and subprograms in such a package must be supplied by a foreign object module.
- Types may not be declared in such a package.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the ALBC30 FOREIGN_BODY command described in Section 4.7.

The pragma is of the form:

pragma FOREIGN_BODY (*language_name* [, *elaboration_routine_name*])

The parameter *language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument provides a means to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object declarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, /CROSS_REFERENCE, (see Section 3.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 3.3).

In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package MATH_FUNCS is
   pragma FOREIGN_BODY ("C");
   function POLYNOMIAL  (X:INTEGER) return INTEGER;
          --Ada spec matching the C routine
   pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
          --Force compiler to use name "plmn" when referring to this
   -- function
end MATH_FUNCS;

with MATH_FUNCS; use MATH_FUNCS
procedure MAIN is
   X:INTEGER := POLYNOMIAL(10);
          -- Will generate a call to "plmn"
   begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCS

2. Compile MAIN

3. Obtain an object module (e.g. math.TOF) containing the compiled code for plmn.

4. Issue the command

    ALBC30 FOREIGN_BODY math_funcs MATH.TOF

5. Issue the command

    ALBC30 LINK MAIN

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command ALBC30 FOREIGN_BODY (See Section 4.7) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

Using Calls to the Operating System. In some cases, the foreign code is actually supplied by the operating system (in the case of system calls) or by runtime libraries for other programming languages such as C. Such calls may be made using a dummy procedure to supply a file specification to the ALBC30 FOREIGN_BODY command. You need a dummy .TOF file which may be obtained in a number of ways. One way is to compile the procedure

```
procedure DUMMY is
begin
    null;
end;
```

Then, use the library command

```
    ALBC30 FOREIGN pkg DUMMY.TOF
```
where *pkg* is the name of the package that contains the pragma LINKAGE_NAME for the operating system call.

For example to use the VMS system service LIB$GET_VM in the program TEST:

```
Package MEMORY is
    pragma FOREIGN_BODY ("ASM");
    procedure GET_VIRTUAL_MEMORY(MEM:INTEGER);
    pragma LINKAGE_NAME (GET_VIRTUAL_MEMORY, "LIB$GET_VM ");
end MEMORY;

with MEMORY;
procedure TEST is
     . . .

begin
     GET_VIRTUAL_MEMORY(MEM);
     . . .
end TEST;
```

Obtain the file dummy.TOF. Then use

```
    ALBC30 FOREIGN pkg DUMMY.TOF
```
to include the body for the system call in the library.

5.2. *IMPLEMENTATION-DEPENDENT ATTRIBUTES*

No implementation-dependent attributes are currently supported.

5.3. *SPECIFICATION OF THE PACKAGE* SYSTEM

The parameter values specified for the TI320C30 target in package SYSTEM [LRM 13.7.1 and Annex C] are:

```
package SYSTEM is
    type ADDRESS is new INTEGER;
    type NAME is (TI320C30);
    SYSTEM_NAME   : constant NAME := TI320C30;
    STORAGE_UNIT  : constant := 32;
    MEMORY_SIZE   : constant := 16_777_216;
    MAX_INT       : constant := 2_147_483_647;
    MIN_INT       : constant := -MAX_INT - 1;
    MAX_DIGITS    : constant := 9;

    MAX_MANTISSA  : constant := 31;
    FINE_DELTA    : constant := 2#1.0#e-31;
    TICK          : constant := 0.0001;
    subtype PRIORITY is INTEGER range 10 .. 100;
    DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
    RUNTIME_ERROR    : exception;
end SYSTEM;
```

5.4. *RESTRICTIONS ON REPRESENTATION CLAUSES*

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. Basic Restriction

The basic restriction on representation specifications [LRM 13.1] that they may be given only for types declared in terms of a type definition, excluding a generic_type_definition (LRM 12.1) and a private_type_definition (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; a diagnostic message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions are not obeyed but cause a diagnostic message to be issued.

5.4.2. Length Clauses

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.

2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   • An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

   ```
   type My_Enum is (A,B);
   for My_enum'size use 1;
   V,W: My_enum;  -- will occupy two storage
                  -- units on the stack
                  -- (if allocated at all)
   type rec is record
      V,W: My_enum;
   end record;
   pragma Pack(rec);
   O: rec;  -- will occupy one storage unit
   ```

   • A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

   • Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational size in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```
type my_int is range 0..65535;
for my_int'size use 16;  -- o.k.
A,B: my_int;
    ...A + B...  -- this operation will generally be
                 -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```
type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
              ...
          X: SMALL_MY_INT;
              ...
      end record;
```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 8 bits.

For the following type classes, the size specification must coincide with the default size chosen by the compiler for the type:

- access types
- floating-point types
- task types

No useful effect can be achieved by using size specifications for these types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the accuracy_definition of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy explained below in adherence to any alignment constraints on the component type (see Section 5.4.7).

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by Pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. Furthermore, some administrative overhead for the allocator must be taken into account by the user (currently 1 word per allocated object).

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is generally no extension of task activations.

5.4.2.7. Specification of ' SMALL

Only powers of 2 are allowed for ' SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of ' SMALL must then be accommodatable within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER' FIRST and INTEGER' LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.

- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5-7

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed. The specified expression must yield a target-dependent value.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless. Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied only to a package if it is a body stub.

- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.

- Specified addresses must be static.

5.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

5.4.6.1. Pragma PACK for Arrays

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to

1. the array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.

2. the component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type string. However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 32-bit word.

5.4.6.3. Pragma PACK for Records

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

    for TOENTRY use at intID;

by associating the interrupt specified by intID with the toentry entry of the task containing this address clause. The interpretation of intID is both machine and compiler dependent.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supplies the predefined input/output packages DIRECT_IO, SEQUENTIAL_IO, TEXT_IO, and LOW_LEVEL_IO as required by LRM Chapter 14. However, since TI320C30 is used in embedded applications lacking both standard I/O devices and file systems, the functionality of DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO is limited.

DIRECT_IO and SEQUENTIAL_IO raise USE_ERROR if a file open or file access is attempted. TEXT_IO is supported to CURRENT_OUTPUT and from CURRENT_INPUT. A routine that takes explicit file names raises USE_ERROR.

5.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the ALBC30 command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada. (Code sharing will be implemented in a later release.)

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will obsolete any units that instantiated this generic unit.

5.9.3. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

    DURATION'DELTA is 0.0001 sec
    DURATION'SMALL is $6.103516E^{-5}$ sec
    DURATION'FIRST is -86400.0 sec
    DURATION'LAST is 86400.0 sec

5.9.4. Values of Integer Attributes

Tartan Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

    INTEGER'FIRST = -2**31
    INTEGER'LAST = 2**31-1

The range bounds for subtypes declared in package TEXT_IO are:

```
COUNT'FIRST = 0
COUNT'LAST = INTEGER'LAST - 1

POSITIVE_COUNT'FIRST = 1
POSITIVE_COUNT'LAST = INTEGER'LAST - 1

FIELD'FIRST = 0
FIELD'LAST = 20
```

The range bounds for subtypes declared in packages DIRECT_IO are:

```
COUNT'FIRST = 0
COUNT'LAST = INTEGER'LAST

POSITIVE_COUNT'FIRST = 1
POSITIVE_COUNT'LAST = COUNT'LAST
```

5.9.5. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types FLOAT and LONG_FLOAT.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	23
EMAX	92
EPSILON approximately	16#0.1000_00#E-4 9.53674E-07
SMALL approximately	16#0.8000_00#E-21 2.58494E-26
LARGE approximately	16#0.FFFF_F8#E+21 1.93428E+25
SAFE_EMAX	126
SAFE_SMALL approximately	16#0.2000_00#E-31 5.87747E-39
SAFE_LARGE approximately	16#0.3FFF_FE#E+32 8.50706E+37
FIRST approximately	-16#0.1000_00#E+33 -3.40282E+38
LAST approximately	16#0.FFFF_FF#E+32 3.40282E+38}
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	128
MACHINE_EMIN	-126
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON approximately	16#0.4000_0000_0#E-7 9.31322575E-10
SMALL approximately	16#0.8000_0000_0#E-31 2.35098870E-38
LARGE approximately SAFE_EMAX	16#0.FFFF_FFFE_0#E+31 2.12676479E+37 1022
SAFE_SMALL approximately	16#0.2000_0000_0#E-31 5.87747175E-39
SAFE_LARGE approximately	16#0.3FFF_FFFF_8#E+32 8.50705917E+37
FIRST approximately	-16#0.1000_0000_0#E+33 -3.40282367E+38
LAST approximately	16#0.FFFF_FFFF_0#E+32 3.40282367E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	32
MACHINE_EMAX	128
MACHINE_EMIN	-126
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

5.10. *SUPPORT FOR PACKAGE MACHINE_CODE*

Package MACHINE_CODE provides the programmer with an interface through which to request the generation of any instruction that is available on the C30. The ADAC30 implementation of package MACHINE_CODE is similar to that described in Section 13.8 of the Ada LRM, with several added features.

5.10.1. Basic Information

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

5.10.2. Instructions

A machine code insert has the form TYPE_MARK' RECORD_AGGREGATE, where the type must be one of the records defined in package MACHINE_CODE. Package MACHINE_CODE defines seven types of records. Each has an opcode and zero to 6 operands. These records are adequate for the expression of all instructions provided by the C30.

5.10.3. Operands

An operand consists of a record aggregate which holds all the information to specify it to the compiler. All operands have an address mode and one or more other pieces of information. The operands correspond exactly to the operands of the instruction being generated.

5.10.3.1. Address Modes

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes provided in package MACHINE_CODE provide access to all address modes supported by the C30.

In addition, package MACHINE_CODE supplies the address modes Symbolic_Address and Symbolic_Value which allow the user to refer to Ada objects by specifying Object'ADDRESS as the value for the operand. Any Ada object which has the 'ADDRESS attribute may be used in a symbolic operand. Symbolic_Address should be used when the operand is a true address (that is, a branch target for example). Symbolic_Value should be used when the operand is actually a value (that is, one of the source operands of an ADDI instruction).

When an Ada object is used as a *source* operand in an instruction (that is, one from which a value is read), the compiler will generate code which fetches the *value* of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler will generate code which uses the *address* of the Ada object as the destination of the instruction.

5.10.4. Examples

The ADAC30 implementation of package MACHINE_CODE makes it possible to specify both simple machine code inserts such as

```
two_format' (LDI, (Imm, 5), (Reg, R0))
```

and more complex inserts such as

```
three_format' (ADDI3,
        (Imm, 10),
        (Symbolic_Value, Array_Var(X, Y, 27)'ADDRESS),
        (Symbolic_Address, Parameter_1'ADDRESS))
```

In the first example, the compiler will emit the instruction LDI 5, R0. In the second example, the compiler will first emit whatever instructions are needed to form the address of Array_Var(X, Y, 27) and then emit the ADDI3 instruction. If Parameter_1 is not found in a register, the compiler will put the result of the addition in a temporary register and then store it to Parameter_1'ADDRESS. Note that the destination operand of the ADDI3 instruction is given as a Symbolic_Address. This holds true for all destination

operands. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the programmer (either through pragma SUPPRESS, or through command qualifiers).

5.10.5. Incorrect Operands

Two modes of operation are supplied for package MACHINE_CODE: Fixup mode and No_Fixup mode. The choice of mode is made with the qualifier /Fixup_Machine_Code or its negation /No_Fixup_Machine_Code.

In No_Fixup mode, the specification of incorrect operands for an instruction is considered to be a fatal error. In this mode, the compiler will not generate any extra instructions to help you to make a machine code insertion. Note that it is still legal to use 'ADDRESS constructs as long as the object which is used meets the requirements of the instruction.

In Fixup mode, if you specify incorrect operands for an instruction, the compiler will do its best to fix up the machine code to provide the desired effect. For example, although it is illegal to use a memory address as the destination of an ADDI instruction, the compiler will accept it and try to generate correct code. In this case, the compiler will load the value found at the memory address indicated into a register, use this register in the ADDI instruction, and then store from that register back to the desired memory location.

```
Two_Opnds'(ADDI, (Imm, 10), (ARI, AR1))
```

will produce a code sequence like

```
LDI      *AR1, R0
ADDI     10, R0
STI      R0, *AR1
```

The next example illustrates the fix up required when the displacement is out of range for the first operand of an ADDI3 instruction. The displacement is first loaded into one of the index registers.

```
Three_Opnds'(ADDI3, (IPDA, AR3, 2), (Reg, R0), (Reg, R1))
```

will produce a code sequence like

```
LDI      2, IR0
ADDI3    AR3(IR0), R0, R1
```

5.10.6. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on your register usage. The compiler will automatically free all the registers which would be volatile across a call for your use (that is, R0..R3, AR0..AR2, IR0, IR1, and DP). If you reference any other register, the compiler will reserve it for your use until the end of the machine code routine. The compiler will *not* save the register automatically if this routine is inline expanded. This means that the first reference to a register which is not volatile across calls should be an instruction which saves its value in a safe place. The value of the register should be restored at the end of the machine code routine. This rule will help ensure correct operation of your machine code insert even if it is inline explaned in another routine. However, the compiler will save the register automatically in the prolog code for the routine and restore it in the epilog code for the routine if the routine is *not* inline expanded.

The compiler may need several registers to generate code for operand fixups in machine code inserts. If you use all the registers, fixups will not be possible. In general, when more registers are available to the compiler it is able to generate better code.

5.10.7. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under programmer control through the use of pragma INLINE, or at Optimization Level 3 when the compiler selects that optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as though it was a call; volatile registers will be saved and restored around it, etc.

5.10.8. Unsafe Assumptions

There are a variety of assumptions which should *not* be made when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or which may not function correctly.

* The compiler *will not* generate call site code for you if you emit a call instruction. You must save and restore any volatile registers which currently have values in them, etc. If the routine you call has out parameters, a large function return result, or an unconstrained result, it is your responsibility to emit the necessary instructions to deal with these constructs as the compiler expects. In other words, when you emit a call, you must follow the linkage conventions of the routine you are calling. For further details on call site code, see Sections 6.4, 6.5 and 6.6.

* Do not assume that the 'ADDRESS on Symbolic_Address or Symbolic_Value operands means that you are getting an ADDRESS to operate on. The Address- or Value-ness of an operand is determined by your choice of Symbolic_Address or Symbolic_Value. This means that to add the *contents* of X to R3, you should write

```
Three_Format'(ADDI, (Symbolic_Value, X'ADDRESS),
                     (Reg, R3))
```

but to add the *address* of X to R3, you should write

```
Three_Format'(ADDI, (Symbolic_Address, X'ADDRESS),
                     (Reg, R3));
```

5.10.9. Package MACHINE_CODE

```
with system;      -- -*- ada -*-
package Machine_Code is

type Instruction_Mnemonic is
    -- the character "i" has been appended to instructions that
    -- conflict with Ada reserved words.
   (ABSF, ABSF_STF, ABSI,
    ABSI_STI, ADDC, ADDC3,
    ADDC3_STI, ADDF, ADDF3,
    ADDF3_STF, ADDI, ADDI3,
    ADDI3_STI, ANDi, AND3,
    AND3_STI, ANDN, ANDN3,
    ASH, ASH3,
    ASH3_STI, BU, BLS,
    BHI, BLO, BHS,
    BEQ, BNE, BLE,
    BGT, BLT, BGE,
    BZ, BNZ, BNP,      -- BNP is another name for BLE
    BP, BN, BNN,
    BNV, BV, BNUF,
    BUF, BNC, BC,
    BNLV, BLV, BNLUF,
    BLUF, BZUF, BUD,
    BLSD, BHID, BLOD,
    BHSD, BEQD, BNED,
    BLED, BGTD, BLTD,
    BGED, BZD, BNZD,
    BNPD, BPD, BND,    -- BNPD is another name for BLED
```

```
        BNND, BNVD, BVD,
        BNUFD, BUFD, BNCD,
        BCD, BNLVD, BLVD,
        BNLUFD, BLUFD, BZUFD,
        BR, BRD, CALL,

        CALL_noret,         -- Another name for CALL.
--                          -- Use this when call never returns.

        CALLU, CALLLS,
        CALLHI, CALLLO, CALLHS,
        CALLEQ, CALLNE, CALLLE,
        CALLGT, CALLLT, CALLGE,
        CALLZ, CALLNZ, CALLNP,    -- CALLNP is another name for CALLLE
        CALLP, CALLN, CALLNN,
        CALLNV, CALLV, CALLNUF,
        CALLUF, CALLNC, CALLC,
        CALLNLV, CALLLV, CALLNLUF,
        CALLLUF, CALLZUF,

        case_jump,                -- Another name for BU.
--                                -- use this when building a CASE statement

        CMPF, CMPF3,
        CMPI, CMPI3,
        DBU, DBLS, DBHI,
        DBLO, DBHS, DBEQ,
        DBNE, DBLE, DBGT,
        DBLT, DBGE, DBZ,
        DBNZ, DBNP, DBP,          -- DBNP is another name for DBLE
        DBN, DBNN, DBNV,
        DBV, DBNUF, DBUF,
        DBNC, DBC, DBNLV,
        DBLV, DBNLUF, DBLUF,
        DBZUF, DBUD, DBLSD,
        DBHID, DBLOD, DBHSD,
        DBEQD, DBNED, DBLED,
        DBGTD, DBLTD, DBGED,
        DBZD, DBNZD, DBNPD,       -- DBNPD is another name for DBLED
        DBPD, DBND, DBNND,
        DBNVD, DBVD, DBNUFD,
        DBUFD, DBNCD, DBCD,
        DBNLVD, DBLVD, DBNLUFD,
        DBLUFD, DBZUFD, FIX,
        FIX_STI, FLOATi, FLOAT_STF,
        IACK, IDLE, LDE,
        LDF, LDFU, LDFLS,
        LDFHI, LDFLO, LDFHS,
        LDFEQ, LDFNE, LDFLE,
        LDFGT, LDFLT, LDFGE,
        LDFZ, LDFNZ, LDFNP,       -- LDFNP is another name for LDFLE
        LDFP, LDFN, LDFNN,
        LDFNV, LDFV, LDFNUF,
        LDFUF, LDFNC, LDFC,
        LDFNLV, LDFLV, LDFNLUF,
        LDFLUF, LDFZUF, LDFI,
        LDFLDF, LDF_STF, LDI,
        LDIU, LDILS, LDIHI,
        LDILO, LDIHS, LDIEQ,
        LDINE, LDILE, LDIGT,
        LDILT, LDIGE, LDIZ,
        LDINZ, LDINP, LDIP,       -- LDINP is another name for LDILE
        LDIN, LDINN, LDINV,
        LDIV, LDINUF, LDIUF,
        LDINC, LDIC, LDINLV,
        LDILV, LDINLUF, LDILUF,
```

```
LDIZUF, LDII, LDILDI,
LDI_STI, LDM, LSH,
LSH3, LSH3_STI, MPYF,
MPYF3, MPYF3_STF,

                    -- MPYF3_ADDF3 srcA, srcB, dst1, srcC, srcD, dst2

MPYF3_ADDF3_1,      -- srcA: indirect; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: Rn Register; dst2 R2 or R3

MPYF3_ADDF3_2,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

MPYF3_ADDF3_3,      -- srcA: Rn Register; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: indirect; srcD: indirect; dst2: R2 or R3

MPYF3_ADDF3_4,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

MPYF3_ADDF3_5,      -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

MPYF3_ADDF3_6,      -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: indirect: srcD: Rn Register; dst2: R2 or R3

                    -- MPYF3_SUBF3 srcA, srcB, dst1, srcC, srcD, dst2

MPYF3_SUBF3_1,      -- srcA: indirect; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: Rn Register; dst2 R2 or R3

MPYF3_SUBF3_2,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

MPYF3_SUBF3_3,      -- srcA: Rn Register; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: indirect; srcD: indirect; dst2: R2 or R3

MPYF3_SUBF3_4,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

MPYF3_SUBF3_5,      -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

MPYF3_SUBF3_6,      -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: indirect: srcD: Rn Register; dst2: R2 or R3

MPYI, MPYI3,

MPYI3_STI,

                    -- MPYI3_ADDI3 srcA, srcB, dst1, srcC, srcD, dst2

MPYI3_ADDI3_1,      -- srcA: indirect; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: Rn Register; dst2 R2 or R3

MPYI3_ADDI3_2,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

MPYI3_ADDI3_3,      -- srcA: Rn Register; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: indirect; srcD: indirect; dst2: R2 or R3

MPYI3_ADDI3_4,      -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                    -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

MPYI3_ADDI3_5,      -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                    -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3
```

```
        MPYI3_ADDI3_6,          -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                                -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

                                -- MPYI3_SUBI3 srcA, srcB, dst1, srcC, srcD, dst2

        MPYI3_SUBI3_1,          -- srcA: indirect; srcB: indirect; dst1: R0 or R1
                                -- srcC: Rn Register; srcD: Rn Register; dst2 R2 or R3

        MPYI3_SUBI3_2,          -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                                -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

        MPYI3_SUBI3_3,          -- srcA: Rn Register; srcB: Rn Register; dst1: R0 or R1
                                -- srcC: indirect; srcD: indirect; dst2: R2 or R3

        MPYI3_SUBI3_4,          -- srcA: Rn Register; srcB: indirect; dst1: R0 or R1
                                -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

        MPYI3_SUBI3_5,          -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                                -- srcC: Rn Register; srcD: indirect; dst2: R2 or R3

        MPYI3_SUBI3_6,          -- srcA: indirect; srcB: Rn Register; dst1: R0 or R1
                                -- srcC: indirect; srcD: Rn Register; dst2: R2 or R3

NEGB, NEGF,
NEGF_STF, NEGI, NEGI_STI,
NOP, NORM, NORM_STF,
NOTi, NOT_STI, ORi,
OR3, OR3_STI, POP,
POPF, PUSH, PUSHF,
RETIU, RETILS, RETIHI,
RETILO, RETIHS, RETIEQ,
RETINE, RETILE, RETIGT,
RETILT, RETIGE, RETIZ,
RETINZ, RETINP, RETIP,        -- RETINP is another name for RETILE
RETIN, RETINN, RETINV,
RETIV, RETINUF, RETIUF,
RETINC, RETIC, RETINLV,
RETILV, RETINLUF, RETILUF,
RETIZUF, RETSU, RETSLS,
RETSHI, RETSLO, RETSHS,
RETSEQ, RETSNE, RETSLE,
RETSGT, RETSLT, RETSGE,
RETSZ, RETSNZ, RETSNP,        -- RETSNP is another name for RETSLE
RETSP, RETSN, RETSNN,
RETSNV, RETSV, RETSNUF,
RETSUF, RETSNC, RETSC,
RETSNLV, RETSLV, RETSNLUF,
RETSLUF, RETSZUF, RND,
ROL,
ROLC, ROR,
RORC,
RPTB, RPTS, SIGI,
STF, STFI, STF_STF,
STI, STII, STI_STI,
SUBB, SUBB3,
SUBC, SUBF, SUBF3,
SUBF3_STF, SUBI, SUBI3,
SUBI3_STI, SUBRB,
SUBRF,
SUBRI,
SWI, TRAPU,
TRAPLS, TRAPHI, TRAPLO,
TRAPHS, TRAPEQ, TRAPNE,
TRAPLE, TRAPGT, TRAPLT,
TRAPGE, TRAPZ, TRAPNZ,
TRAPNP, TRAPP, TRAPN,         -- TRAPNP is another name for TRAPLE
```

```
        TRAPNN, TRAPNV, TRAPV,
        TRAPNUF, TRAPUF, TRAPNC,
        TRAPC, TRAPNLV, TRAPLV,
        TRAPNLUF, TRAPLUF, TRAPZUF,
        TSTB, TSTB3,
        XORi, XOR3, XOR3_STI);

    type Register is
        ( R0,   R1,   R2,   R3,   R4,   R5,   R6,   R7,
        Ar0, AR1, AR2, AR3, AR4, AR5, AR6, AR7,
        DP,
        IR0, IR1,
        BK, SP, ST, IE,
        IFlags, -- expanded name to avoid conflict with ADA reserved word "IF"
        IOF,
        RS, RE, RC);


    type FloatLit is digits 9;


    subtype Auxiliary_Register        is Register range AR0 .. AR7;
    subtype Index_Register            is Register range IR0 .. IR1;
    subtype Data_Page_Pointer         is Register range DP  .. DP;

    subtype Data_Page_Offset          is integer range 0 .. 16777215;
    subtype Displacement              is integer range 0 .. 255;

    type Address_Mode is
        (Imm,       -- short Immediate
        LongImm,  -- longe Immediate
        FloatImm, -- short float immediate
        Direct,   -- Direct
        Reg,      -- Register
        ARI,      -- Auxiliary Register Indirect
        IPDA,     -- Indirect with Pre -Displacement Add
        IPDS,     -- Indirect with Pre -Displacement Subtract
        IPrDAM,   -- Indirect with Pre -Displacement Add        & Modify
        IPrDSM,   -- Indirect with Pre -Displacement Subtract & Modify
        IPoDAM,   -- Indirect with Post-Displacement Add        & Modify
        IPoDSM,   -- Indirect with Post-Displacement Subtract & Modify
        IPoDACM,  -- Indirect with Post-Displacement Add        & Circular Modify
        IPoDSCM,  -- Indirect with Post-Displacement Subtract & Circular Modify
        IPrIA,    -- Indirect with Pre -Index     Add
        IPrIS,    -- Indirect with Pre -Index     Subtract
        IPrIAM,   -- Indirect with Pre -Index     Add        & Modify
        IPrISM,   -- Indirect with Pre -Index     Subtract & Modify
        IPoIAM,   -- Indirect with Post-Index     Add        & Modify
        IPoISM,   -- Indirect with Post-Index     Subtract & Modify
        IPoIACM,  -- Indirect with Post-Index     Add        & Circular Modify
        IPoISCM,  -- Indirect with Post-Index     Subtract & Circular Modify
        IPoIABRM, -- Indirect with Post-Index     Add        & Bit-Reverse Modify
        PcRel,    -- Pc-relative (a label, for example)

        Symbolic_Address, -- special use to express the address of an ADA object
        Symbolic_Value);  -- special use to express the value   of an ADA object

    type Operand(AM : Address_Mode := Imm -- defaulted to keep Ada Happy
            ) is
        record
            -- This record describes all the possible representations of an operand.
            -- It must be coordinated with the code generator's model.
            case AM is
              when Imm =>
                    -- Short Immediate
                    K: Integer;

              when LongImm =>
```

```
              -- longe Immediate
              Address : System.Address;

       when FloatImm =>
              -- Short Float Immediate
              F: FloatLit;

       when Direct =>
              -- Direct
              Page_Address : Data_Page_Offset;

       when Reg =>
              -- Register
              R_Reg : Register;

       when ARI =>
              -- Auxiliary Register Indirect
              ARI_Ar : Auxiliary_Register;

       when IPDA =>
              -- Indirect with Pre-Displacement Add
              IPDA_Ar    : Auxiliary_Register;
              IPDA_disp : Displacement;

       when IPDS =>
              -- Indirect with Pre-Displacement Subtract
              IPDS_Ar    : Auxiliary_Register;
              IPDS_disp : Displacement;

       when IPrDAM =>
              -- Indirect with Pre-Displacement Add & Modify
              IPrDAM_Ar    : Auxiliary_Register;
              IPrDAM_disp : Displacement;

       when IPrDSM =>
              -- Indirect with Pre-Displacement Subtract & Modify
              IPrDSM_Ar    : Auxiliary_Register;
              IPrDSM_disp : Displacement;

       when IPoDAM =>
              -- Indirect with Post-Displacement Add & Modify
              IPoDAM_Ar    : Auxiliary_Register;
              IPoDAM_disp : Displacement;

       when IPoDSM =>
              -- Indirect with Post-Displacement Subtract & Modify
              IPoDSM_Ar    : Auxiliary_Register;
              IPoDSM_disp : Displacement;

       when IPoDACM =>
              -- Indirect with Post-Displacement Add & Circular Modify
              IPoDACM_Ar    : Auxiliary_Register;
              IPoDACM_disp : Displacement;

       when IPoDSCM =>
              -- Indirect with Post-Displacement Subtract & Circular Modify
              IPoDSCM_Ar    : Auxiliary_Register;
              IPoDSCM_disp : Displacement;

       when IPrIA =>
              -- Indirect with Pre-Index Add
              IPrIA_Ar : Auxiliary_Register;
              IPrIA_Ir : Index_Register;


       when IPrIS =>
```

```
                        -- Indirect with Pre-Index Subtract
                        IPrIS_Ar : Auxiliary_Register;
                        IPrIS_Ir : Index_Register;

                  when IPrIAM =>
                        -- Indirect with Pre-Index Add & Modify
                        IPrIAM_Ar : Auxiliary_Register;
                        IPrIAM_Ir : Index_Register;

                  when IPrISM =>
                        -- Indirect with Pre-Index Subtract & Modify
                        IPrISM_Ar : Auxiliary_Register;
                        IPrISM_Ir : Index_Register;

                  when IPoIAM =>
                        -- Indirect with Post-Index Add & Modify
                        IPoIAM_Ar : Auxiliary_Register;
                        IPoIAM_Ir : Index_Register;

                  when IPoISM =>
                        -- Indirect with Post-Index Subtract & Modify
                        IPoISM_Ar : Auxiliary_Register;
                        IPoISM_Ir : Index_Register;

                  when IPoIACM =>
                        -- Indirect with Post-Index Add & Circular Modify
                        IPoIACM_Ar : Auxiliary_Register;
                        IPoIACM_Ir : Index_Register;

                  when IPoISCM =>
                        -- Indirect with Post-Index Subtract & Circular Modify
                        IPoISCM_Ar : Auxiliary_Register;
                        IPoISCM_Ir : Index_Register;

                  when IPoIABRM =>
                        -- Indirect with Post-Index Add & Bit-Reverse Modify
                        IPoIABRM_Ar : Auxiliary_Register;
                        IPoIABRM_Ir : Index_Register;

                  when PcRel =>
                        -- Pc-relative (a label, for example)
                        L_Addr: System.Address; -- 'ADDRESS of an Ada label

                  when Symbolic_Address =>
                        -- special use to express the address of an ADA object
                        S_Addr: System.Address;

                  when Symbolic_Value =>
                        -- special use to express the value of an ADA object
                        S_Value: System.Address;

            end case;
      end record;


type Zero_Opnds is
      Record
            Operation: Instruction_Mnemonic;
      end Record;

type One_Opnds is
      Record
            Operation: Instruction_Mnemonic;
            Operand_1: Operand;
      end Record;
```

```
    type Two_Opnds is
       Record
          Operation: Instruction_Mnemonic;
          Operand_1: Operand;
          Operand_2: Operand;
       end Record;

    type Three_Opnds is
       Record
          Operation: Instruction_Mnemonic;
          Operand_1: Operand;
          Operand_2: Operand;
          Operand_3: Operand;
       end Record;

    type Four_Opnds is
       Record
          Operation: Instruction_Mnemonic;
          Operand_1: Operand;
          Operand_2: Operand;
          Operand_3: Operand;
          Operand_4: Operand;
       end Record;

    type Five_Opnds is
       Record
          Operation: Instruction_Mnemonic;
          Operand_1: Operand;
          Operand_2: Operand;
          Operand_3: Operand;
          Operand_4: Operand;
          Operand_5: Operand;
       end Record;

    type Six_Opnds is
       Record
          Operation: Instruction_Mnemonic;
          Operand_1: Operand;
          Operand_2: Operand;
          Operand_3: Operand;
          Operand_4: Operand;
          Operand_5: Operand;
          Operand_6: Operand;
       end Record;


  end Machine_Code;
```

5.10.10. Example

```
with machine_code; use machine_code;

procedure mach_example is

      type ary_type is new string(1..5);

      st1,st2: ary_type;

      procedure move_string(length: in integer; s1: in ary_type;
                            s2: out ary_type) is
      -- move string s1, which is length characters, to string s2
      -- if this routine was inline expanded, instructions should be present
      -- to save and restore any non-volatile registers.
      begin

         Two_Opnds'(LDI, (Symbolic_Value, length'Address), (Reg, R3));
```

```
            Two_Opnds'(LDI, (Symbolic_Address, s1'Address), (Reg, Ar0));
            Two_Opnds'(LDI, (Symbolic_Address, s2'Address), (Reg, Ar1));
            Two_Opnds'(LDI, (IPoDAM, Ar0, 1), (Reg, R4));
            One_Opnds'(RPTS, (Reg, R3));
            Four_Opnds'(LDI_STI, (IPoDAM, Ar0, 1), (Reg, R4),
                    (Reg, R4), (IPoDAM, Ar1, 1));
            Two_Opnds'(STI, (Reg, R4), (IPoDAM, Ar1, 1));

    end move_string;

begin

    st1 := "hello";
    move_string(5, st1, st2);

end mach_example;
```

Assembly code output:

```
        .global mach_example
; mach_example.tmp from src/mach_example.ada
; Ada Sun/C30 Version V11.466293001 Copyright 1989, Tartan Laboratories
; 1986
        .global xxml002
        .ref    ARTSTACKPROLOG

        .text

xxml002:        PUSH    AR3
        LDI     SP,AR3
        PUSH    AR3
        LDI     20,AR0
        CALL    ARTSTACKPROLOG
        ADDI    14,SP
        PUSH    RS
        PUSH    RE
        PUSH    RC
        LDI     @DEF1,AR0
        STI     AR0,*+AR3(1)
        LDI     1,R1
        STI     R1,*+AR3(2)
        STI     R1,*+AR3(3)
        LDI     5,R0
        STI     R0,*+AR3(4)
        STI     R0,*+AR3(5)             ; line 5
        LDI     3,RC            ; line 26
        LDI     @DEF2,AR0
        LDI     AR3,AR1
        ADDI    6,AR1
        LDI     *AR0++(1),R2
        RPTB    L8
L8:     LDI     *AR0++(1),R2
        || STI  R2,*AR1++(1)
        STI     R2,*AR1++(1)
        LDI     5,R0            ; line 27
        LDI     AR3,AR0
        ADDI    2,AR0
        LDI     AR3,AR1
        ADDI    2,AR1
        LDI     4,R1
L9:     LDI     *AR0++(1),R2
        CMPI    *AR1++(1),R2
        BEQ     L10
        LDI     0,R1
        BR      L12
L10:    SUBI    1,R1
        BGT     L9
        LDI     1,R1
L12:    CMPI    1,R1
        BNE     L13
        LDI     AR3,R1
        ADDI    6,R1
        LDI     AR3,AR0
        ADDI    2,AR0
        LDI     AR3,AR1
        ADDI    2,AR1
        LDI     4,R2
L14:    LDI     *AR0++(1),R3
        CMPI    *AR1++(1),R3
        BEQ     L15
        LDI     0,R2
        BR      L17
L15:    SUBI    1,R2
```

```
            BGT     L14
            LDI     1,R2
L17:        CMPI    1,R2
            BNE     L13
            LDI     AR3,R2
            ADDI    11,R2
            CALL    move_string$00
L7:
            LDI     *+AR3(16),RS
            LDI     *+AR3(17),RE
            LDI     *+AR3(18),RC
            LDI     AR3,SP
            POP     AR3
            RETSU
L13:        TRAPU   15

; Total words of code in the above routine = 68

            .data
DEF1:       .word   L7


            .text

move_string$00: PUSH    AR3
            LDI     SP,AR3
            PUSH    AR3
            PUSH    R4
            PUSH    R6
            LDI     @DEF3,AR0
            STI     AR0,*+AR3(1)
            LDI     R0,R3           ; line 13
            LDI     R1,AR0          ; line 14
            LDI     R2,AR1          ; line 15
            LDI     *AR0++(1),R4            ; line 16
            RPTS    R3              ; line 17
            LDI     *AR0++(1),R4
         || STI     R4,*AR1++(1)             ; line 18
            STI     R4,*AR1++(1)             ; line 20
            POP     R2              ; line 13
            POP     R1
            POP     R0
L19:
            LDI     *+AR3(2),R4
            LDI     *+AR3(3),R6
            LDI     AR3,SP
            POP     AR3
            RETSU

; Total words of code in the above routine = 22

            .data
DEF3:       .word   L19


            .text
            .data
DEF2:       .word   DEF4
DEF4:       .byte   104,101 ; he
            .byte   108,108 ; ll
            .byte   111     ; o

; Total words of code = 90
; Total words of data = 8

            .end
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below. The use of the '*' operator signifies a multiplication of the following character, and the use of the '&' character signifies concatenation of the preceeding and following strings. The values within single or double quotation marks are to highlight character or string values:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	239 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	239 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	120 * 'A' & '3' & 119 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	120 * 'A' & '4' & 119 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	237 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	235 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 120 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 119 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	220 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	16777216
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	32

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TI320C30
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	20
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	100
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/NON_EXISTENT_DIRECTORY1/FILE1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NON_EXISTENT_DIRECTORY2/FILE2

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	10
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	240
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

Name and Meaning   Value

$MAX_LEN_INT_BASED_LITERAL   "2:" & 235 * '0' & "11:"
    A   universal   integer   based
    literal whose value is 2#11#
    with enough leading zeroes in
    the mantissa to be MAX_IN_LEN
    long.

$MAX_LEN_REAL_BASED_LITERAL   "16:" & 233 * '0' & "F.E:"
    A universal real based literal
    whose   value   is   16:F.E: with
    enough leading zeroes in the
    mantissa to be MAX_IN_LEN long.

$MAX_STRING_LITERAL   '"' & 238 * 'A' & '"'
    A   string   literal   of   size
    MAX_IN_LEN, including the quote
    characters.

$MIN_INT   -2147483648
    A   universal   integer   literal
    whose value is SYSTEM.MIN_INT.

$MIN_TASK_SIZE   32
    An integer literal whose value
    is the number of bits required
    to hold a task object which has
    no entries, no declarations, and
    "NULL;" as the only statement in
    its body.

$NAME   NO_SUCH_TYPE_AVAILABLE
    A name of a predefined numeric
    type other than FLOAT, INTEGER,
    SHORT_FLOAT,   SHORT_INTEGER,
    LONG_FLOAT,   or   LONG_INTEGER.

$NAME_LIST   TI320C30
    A list of enumeration literals
    in   the   type   SYSTEM.NAME,
    separated by commas.

$NEG_BASED_INT   16#FFFFFFFE#
    A based integer literal whose
    highest   order   nonzero   bit
    falls   in   the   sign   bit
    position of the representation
    for SYSTEM.MAX_INT.

Name and Meaning   Value

$NEW_MEM_SIZE   16777216
    An integer literal whose value
    is a permitted argument for
    pragma MEMORY_SIZE, other than
    $DEFAULT_MEM_SIZE. If there is
    no other value, then use
    $DEFAULT_MEM_SIZE.

$NEW_STOR_UNIT   32
    An integer literal whose value
    is a permitted argument for
    pragma STORAGE_UNIT, other than
    $DEFAULT_STOR_UNIT. If there is
    no other permitted value, then
    use value of SYSTEM.STORAGE_UNIT.

$NEW_SYS_NAME   TI320C30
    A value of the type SYSTEM.NAME,
    other than $DEFAULT_SYS_NAME. If
    there is only one value of that
    type, then use that value.

$TASK_SIZE   96
    An integer literal whose value
    is the number of bits required
    to hold a task object which has
    a single entry with one 'IN OUT'
    parameter.

$TICK   0.0001
    A real literal whose value is
    SYSTEM.TICK.

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C   This test expects that the string "-- TOP OF PAGE.   -- 63" of line 204 will   appear at the top of the listing page   due to a pragma PAGE in line 203; but   line 203 contains text   that follows the pragma, and it is this that must   appear at the   top of the page.

b. A39005G   This test unreasonably expects a component   clause   to pack an array component   into a minimum size (line 30).

c. B97102E   This test contains an unitended illegality: a   select statement contains a   null statement at the place of a selective wait alternative (line 31).

d. C97116A   This test contains race conditions, and it assumes that guards are evaluated   indivisibly. A conforming   implementation may use interleaved execution in   such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED   at   one   of lines 52 or 56.

e. BC3009B   This test wrongly expects that circular   instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on;   by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D   This test wrongly requires that an array object's size be no greater than 10   although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]   These tests wrongly attempt to check the size of objects of a   derived type   (for which a 'SIZE length clause is given) by passing them to a   derived subprogram (which implicitly converts them to the parent type (Ada standard   3.4:14)). Additionally, they use the 'SIZE length clause and attribute,   whose   interpretation   is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests]   These tests
    assume that dependent tasks will terminate while the main pro-
    gram executes a loop that simply tests for task termination; this
    is not   the case, and the main program may loop indefinitely
    (lines 74, 85, 86 & 96,   86 & 96, and 58, resp.).

i. CD2B15C & CD7205C   These tests expect that a 'STORAGE_SIZE
    length clause provides precise control over the number of
    designated objects in a collection; the Ada standard   13.2:15
    allows that such control must not be expected.

j. CD2D11B   This test gives a SMALL representation clause for a
    derived fixed-point type   (at line 30) that defines a set of
    model numbers that are not necessarily   represented in the
    parent type; by Commentary AI-00099, all model numbers of   a
    derived fixed-point type must be representable values of the
    parent type.

k. CD5007B   This test wrongly expects an implicitly declared sub-
    program to be at the   the address that is specified for an un-
    related subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests]   These tests check
    various aspects of the use of the three SYSTEM pragmas;   the AVO
    withdraws these tests as being inappropriate for validation.

m. CD7105A   This test requires that successive calls to CALENDAR.-
    CLOCK change by at   least SYSTEM.TICK; however, by Commentary
    AI-00201, it is only the expected   frequency of change that must
    be at least SYSTEM.TICK--particular instances   of change may be
    less (line 29).

n. CD7203B, & CD7204B   These tests use the 'SIZE length clause and
    attribute, whose interpretation   is considered problematic by
    the WG9 ARG.

o. CD7205D   This test checks an invalid test objective: it treats
    the specification of   storage to be reserved for a task's
    activation as though it were like the   specification of storage
    for a collection.

p. CE2107I   This test requires that objects of two similar scalar
    types be distinguished   when read from a file--DATA_ERROR is
    expected to be raised by an attempt to   read one object as of
    the other type. However, it is not clear exactly how   the Ada
    standard 14.2.4:4 is to be interpreted; thus, this test objective
    is   not considered valid. (line 90)

q. CE3111C   This test requires certain behavior, when two files are
    associated with the   same external file, that is not required by

the Ada standard.

r. CE3301A   This test contains several calls to END_OF_LINE &
    END_OF_PAGE that have no   parameter: these calls were intended
    to specify a file, not to refer to   STANDARD_INPUT (lines 103,
    107, 118, 132, & 136).

s. CE3411B   This test requires that a text file's column number be
    set to COUNT'LAST in   order to check that LAYOUT_ERROR is raised
    by a subsequent PUT operation.   But the former operation will
    generally raise an exception due to a lack of   available disk
    space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

Chapter 3
Compiling Ada Programs

The ADAC30 command is used to compile and assemble Ada compilation units.

3.1. THE ADAC30 COMMAND FORMAT

The ADAC30 command has this format:

ADAC30 [/qualifier [(option, . . .) . . .]] file-spec [/qualifier [(option, . . .) . . .]]

The parameter file-spec is a source file name. Since the source files need not reside in the directory in which the compilation takes place, file-spec must include sufficient directory information to locate the file. If no extension is supplied with the file name, a default extension of .ADA will be supplied by the compiler.

ADAC30 will accept only one source file per compilation. The source file may contain more than one compilation unit, but it is considered good practice to place only one compilation unit in a file. The compiler sequentially processes all compilation units in the file. Upon successful compilation of a unit,

* the Ada program library ADA.DB is updated to reflect the new compilation time and any new dependencies
* one or more separate compilation files and/or object files are generated

If no errors are detected in a compilation unit, The compiler produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced, and the library is updated.

3.2. COMMAND QUALIFIERS

Command qualifiers indicate special actions to be performed by the compiler or special output file properties. The following qualifiers are available:

/CROSS_REFERENCE
/NOCROSS_REFERENCE [default]
> Controls whether the compiler generates a cross-reference table of linknames for the compilation unit. The table will be placed in the file unit-name.XRF (See Section 3.5).

/ERROR_LIMIT=n
> Stop compilation and produce a listing after n errors are encountered, where n is in the range 0..254. The /ERROR_LIMIT qualifier cannot be negated.

/FIXUP_MACHINE_CODE [default]
/NOFIXUP_MACHINE_CODE
> When package MACHINE_CODE is used, controls whether the compiler attempts to alter operand address modes when those address modes are used incorrectly. When /FIXUP_MACHINE_CODE is specified on the command line, the compiler will possibly generate extra instructions to fix incorrect addressing modes in the machine code insertion. The compiler will be able to "fixup" most incorrectly

specified operands; however, certain illegal instruction/operand combinations may cause the compiler to crash. When /NOFIXUP_MACHINE_CODE is used, the specification of incorrect operands for an instruction is considered to be an error. In this case, no code is generated and a listing is produced.

/LIST[=option]
/NOLIST

Controls whether a listing file is produced. If produced, the file has the source file name and a .LIS extension. The available options are:

ALWAYS   Always produce a listing file
NEVER   Never produce a listing file, equivalent to /NOLIST
ERROR   Produce a listing file only if a compilation error or warning occurs

When no form of this qualifier is supplied in the command line, the default condition is /LIST=ERROR. When the LIST qualifier is supplied without an option, the default option is ALWAYS.

/MACHINE_CODE
/NOMACHINE_CODE [default]

Controls whether the assembly code files produced by the compiler are retained in the user's directory after compilation is complete. This qualifier is useful if the user wishes to inspect the compiler output for code correctness and quality. The default is /NOMACHINE which deletes these machine language files.

/NOENUMIMAGE

Controls whether compiler omits data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the 'IMAGE attribute. You should use /NOENUMIMAGE only when you can guarantee that no unit that will import the enumeration type will use 'IMAGE. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this qualifier is not needed. The compiler can recognize when 'IMAGE is not used and will not generate the supporting strings. The /NOENUMIMAGE qualifier cannot be negated.

/OPT=n

Controls the level of optimization performed by the compiler, requested by n. The /OPT qualifier cannot be negated. The optimization levels available are:

$n = 0$ **Minimum** - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis.

$n = 1$ **Low** - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order.

$n = 2$ **Space** - *This is the default level if none is specified.* Performs level 1 optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. Level 2 also performs lifetime analysis to improve register allocation. It also performs inline expansion of subprogram calls indicated by Pragma INLINE which appears in the same compilation unit.

$n = 3$ Time - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.

/PHASES
/NOPHASES [default]   Controls whether the compiler announces each phase of processing as it occurs.

/SUPPRESS[=(option, ...)]

Suppresses the specific checks identified by the options supplied. The parentheses may be omitted if only one option is supplied. The /SUPPRESS qualifier has the same effect as a global pragma SUPPRESS applied to the source file. If the source program also contains a pragma SUPPRESS, then a given check is suppressed if either the pragma or the qualifier specifies it; that is, the effect of a pragma SUPPRESS cannot be negated with the command line qualifier. The /SUPPRESS qualifier cannot be negated.

The available options are:

ALL	Suppress all checks. This is the default if the qualifier is supplied with no option.
ACCESS_CHECK	As specified in the Ada LRM, Section 11.7.
CONSTRAINT_CHECK	Equivalent of (ACCESS_CHECK, INDEX_CHECK, DISCRIMINANT_CHECK, LENGTH_CHECK, RANGE_CHECK).
DISCRIMINANT_CHECK	As specified in the Ada LRM, Section 11.7.
DIVISION_CHECK	Will suppress compile-time checks for division by zero, but the hardware does not permit efficient run-time checks, so none are done.
ELABORATION_CHECK	As specified in the Ada LRM, Section 11.7.
INDEX_CHECK	As specified in the Ada LRM, Section 11.7.
LENGTH_CHECK	As specified in the Ada LRM, Section 11.7.
OVERFLOW_CHECK	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.
RANGE_CHECK	As specified in the Ada LRM, Section 11.7.
STORAGE_CHECK	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the allocator as a result of a new operation.

/WARNINGS [default]
/NOWARNINGS   Controls whether the warning messages generated by the compiler are displayed to the user at the terminal and in a listing file, if produced.

3.3. WHAT UPDATES ARE MADE TO THE PROGRAM LIBRARY

Simply stated, upon successful compilation of a unit,

- the Ada program library ADA.DB is updated to reflect the new compilation time and any new dependencies
- one or more separate compilation files and/or object files are generated.

However, more complicated situations can arise. The following items list the types of compilation units and address the range of situations that can arise.

- In all cases the transitive closure of the dependencies of a compilation unit in the library must be consistent; that is, the unit must be compiled consistently as defined in section 10.3 of the LRM. A secondary unit can have the specification of its ancestor in its context clause, although it is redundant. For a more complete discussion of closure, see Section 4.5.

- A package specification replaces any library unit in the library with the same name, or is simply added if no such library unit exists.

- A package body replaces any existing body of a package specification with the same name. If no such specification exists, an error message is issued. If such a specification exists, but the body does not match the specification in the sense of Section 7.1 of the LRM, error messages are issued.

- A subprogram specification replaces any library unit in the library with the same name, or is simply added if no such library unit exists.

- A subprogram body replaces any existing body of a (generic or non-generic) subprogram specification with the same name. If no such specification exists, an implicit specification is derived from the body and entered into the library. If such a specification exists, but the body does not match the specification in the sense of Section 6.3 of the LRM, error messages are issued. If any library unit other than a subprogram specification exists with the same name, the new implicit specification replaces that library unit.

- Generic package specifications and subprogram specifications act as explicit specifications, i.e., corresponding bodies must match their specifications. If a generic unit is instantiated, a dependency is created on the generic body.

- Generic instances compiled as library units are treated in the same way as their non-generic counterparts.

- When an instantiation replaces the body of a library unit, all secondary units of that library unit are now obsolete and are deleted.

- A subunit with a parent unit containing an appropriate body stub existing in the library replaces any subunit with the same subunit name, comprised of the stub's name and the name of the ancestor unit, or is simply added, if no such subunit exists. A unit containing stubs will only be entered into the library if the simple names of all its stubs are unique for all stubs derived from its common ancestor. An error message is issued if no parent unit exists in the library, the parent unit exists but does not have a relevent stub, or the parent unit and body stub exist but the subunit does not match the stub or its specification.

- When the parent unit of a subunit is recompiled and the parent no longer contains a stub for the subunit, the subunit, which is now obsolete, is deleted.

3.4. FILES PRODUCED OR USED BY THE COMPILER

Files with the following extensions are contained in the standard packages directories or can be created by compiling or linking an Ada program; the file name is the name of a compilation unit, but may be compressed to conform to length limitations of VMS file management:

ASM	The assembly language file produced by compiling an Ada unit body with /MACHINE_CODE.
BOD	Representation of the body of a generic, and/or the visibility information available to any subunits. *body-name*.BOD is read when compiling a program that instantiates *body-name*, or is a subunit of *body-name*.

DI	Representation of a unit specification. *unit-name*. DI is read during the compilation of a program that does a "with *unit-name*".
LIS	A listing produced by the Ada compiler.
STOF	The object file produced by compiling an Ada unit specification with /MACHINE_CODE.
TOF	The object file created by compiling an ADA unit body.
XRF	Cross reference files that relate Ada names with compressed and disambiguated names used in the assembly language file.
XTOF	Executable image produced by the ALBC30 LINK command.

The following are the extensions used for files that are created temporarily during the linking/compiling process:

ETOF	Object file containing calls for elaboration of the program
TMP	Temporary file used when creating .ASM files

Additionally, temporary files are created during compilation that have the same file extensions listed above, but also have a unique 8 digit hexadecimal number concatenated to the extension. These files will appear in the directory only if a link or compilation is abnormally terminated, and should be deleted by the user.

Files having the following extensions are controlled by the librarian and compiler: DI, BOD, STOF, and TOF. The user should not use these extensions for any other purpose. The ALBC30 DELETE command will automatically delete these files when the respective unit is deleted from the library. If the user deletes these files in any other way, subsequent invocations of the compiler or librarian will have unpredictable results, including fatal crashes. We therefore advise that the user *never* delete these files by operating system commands.

3.5. *THE CROSS REFERENCE MAP OF LINKNAMES*

A cross reference of symbolic names to linknames is generated with the qualifier /CROSS_REFERENCE to the ADAC30 command. The cross-reference file has the extension .XRF; the file name is that of the compiled unit, but possibly compressed to match VMS restrictions, as shown in the example below.

For longer unit-names, the cross reference file generated may not have an immediately obvious name, in order to find it, it may be necessary to search the current working directory for a recently produced file with extention XRF.

Example:
File CREXAMPLE_SPEC.ADA

```
    package THIS_IS_A_LONG_PACKAGE_NAME is
        package ANOTHER_LONG_PACKAGE_NAME is
            procedure LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME;
        end ANOTHER_LONG_PACKAGE_NAME;
    end THIS_IS_A_LONG_PACKAGE_NAME;
```

File CREXAMPLE_BODY.ADA

```
    package body THIS_IS_A_LONG_PACKAGE_NAME is
        package body ANOTHER_LONG_PACKAGE_NAME is separate;
    end THIS_IS_A_LONG_PACKAGE_NAME;
```

File CREXAMPLE.SEP

```
   separate (THIS_IS_A_LONG_PACKAGE_NAME)
   package body ANOTHER_LONG_PACKAGE_NAME is
     procedure LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME is
       begin
         null;
       end LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME;
   end ANOTHER_LONG_PACKAGE_NAME;
```

The commands:

```
   $ ADAC30 CREXAMPLE_SPEC.ADA
   $ ADAC30 CREXAMPLE_BODY.ADA
   $ ADAC30/CROSS_REFERENCE CREXAMPLE.SEP
```

produce the file `THSSLNGPCKGNMSNTHRLNGPCKGNME001.XRF` which appears below:

```
---- Linkname Cross Reference Map thsslngpckgnmSnthrlngpckgnme001.xrf ----

this_is_a_long_package_name=>xxthsskgnm001 at          0
this_is_a_long_package_nameYthis_is_a_long_package_name=>this_is_a_long
_package_nameY00 at         1
this_is_a_long_package_nameYanother_long_package_nameYanother_long_package
_name=>another_long_package_nameY00 at         2
this_is_a_long_package_nameYanother_long_package_nameYlong_procedure_name
_that_will_have_short_linkname=>xxthsskgnm001YlngprcdtlnknmY00 at          3
```

In the above cross reference file:

- The first line represents the name for the elaboration code for the package `THIS_IS_A_LONG_PACKAGE_NAME`. The symbols representing the specification and body have respectively `YDECLARE` and `YBODY` postpended.

- The second line is the name of the data segment for the package `THIS_IS_A_LONG_PACKAGE_NAME`.

- The third line is the name of the data segment for the package `ANOTHER_LONG_PACKAGE_NAME`.

- The name for the `LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME` procedure is on the fourth line.

- The fifth line is the name for the elaboration variable for the procedure `LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME`. Ada rules require that the body of a subprogram is already elaborated before it can be called. If it is not already elaborated the exception `PROGRAM_ERROR` must be raised. For each subprogram that may require an elaboration check the compiler generates a variable that is used to record that the body of the subprogram has been elaborated. The name of the elaboration variable is generated by postpending `YGOTO` to the name of the subprogram. The elaboration variable name is then subject to the same compression algorithms as the rest of the symbols in the program.

The VMS command `DIR/DATE *.XRF` will help you locate the cross reference listing. In order to view the contents of the cross reference file, make sure that your terminal is set to wrap around mode. The identifier appearing at the left is the identifier that appears in the Ada source code. The name to the right of the '=>' is the linkname that is supplied for that identifier to the Tartan Linker. The "at *<number>*" gives the line number in the source code where the identifier is found. A Cross Reference Map can be used to verify that there are no conflicting linknames in a program library that uses subprograms written in another language (see Section 5.1.2.2 that discusses the pragma `FOREIGN_BODY`). It is also useful for assembly-level debugging.

3.6. COMPILER AND LIBRARY DIAGNOSTIC MESSAGES

The compiler issues *diagnostic messages* that appear at your terminal and in the optional compiler-generated listing. Most messages issued by the Tartan Ada Compiler contain a reference to the Ada LRM section and paragraph relevant to the error. This section explains the kinds of diagnostic messages the compiler generates, how the compiler attempts to deal with problems that caused the messages and how you should go about correcting a program.

A comprehensive listing of all the messages the compiler can issue is contained in Appendix Section A.1. Messages in the range 6000 to 6999 are issued by the *library*, not the compiler.

3.6.1. Message Severity Levels

Every message issued by the compiler is assigned a *severity level* that indicates how serious the problem is. There are four message categories.

1. A *fatal error* is serious enough to suspend compilation immediately after the error is discovered. This is the only class of error that inhibits further analysis of the source program. An example of a fatal error message is:

```
Fatal 6801: <library administration file name> is incompatible
    with this version of the library.
```

2. An *error* is serious enough to prevent the generation of object code, but the compiler attempts to recover from the error and continues checking the source for additional errors. An example of an error is:

```
Error 2060: This record field has already been assigned
    in the aggregate (4.3 (6))
```

3. A *warning* does not stop the compiler from generating object code, but may still be an indication of a programming error. When a warning occurs, the code generated may not be what you intended. An example of a warning is:

```
Warn 4001: Elaboration of this subtype will raise
    constraint_error at runtime (3.3.2).
```

4. An *informational* message provides you with additional information when you use some library commands. See, for example, Section 4.6. Informational messages are not issued by the compiler. An example of an informational message is:

```
Info 6011: The files required for linking by <unit_kind>
    <unit_name> are consistent (10.3).
```

3.6.2. Message Formats

The format of messages appearing on the standard error output and in the listing file is similar. Here is an example:

```
5 |         s1 : string(1 .. discrim);
6 |         s2 : string(1 .. 2 * discrim);
                                  ^1
***  1 Error 2204: A discriminant may not be used in this
***        expression (3.7.1 (6))
    7 |   end record;
```

The numbered lines in the example are lines from the source program. The source line in question is followed by the messages and pointers to the exact location of the problem.

On the terminal, horizontal lines are used to separate messages coming from different parts of the source program, for example:

```
   5|          s1 : string(1 .. discrim);
   6|          s2 : string(1 .. 2 * discrim);
                                  ^1
*** 1 Error 2204: A discriminant may not be used
***     in this expression (3.7.1 (6))
   7|    end record;
```

```
  14|          null;
  15|    when numeric_error | constraint_error =>
                               ^1
*** 1 Error 3112: A given exception may only appear once
***     in a handler (11.2 (5))
  16|          null;
```

In a listing file, a *message chain* accompanies each diagnostic message. The message chain indicates where in the program the next and previous messages occur, for example:

```
Ada C30 Beta 1.0  Copyright 1989, Tartan Laboratories Incorporated

*** First diagnostic is on line 6

   1|procedure sample_program is
   2|    subtype small_int is integer range 1 .. 10;
   3|
   4|    type rec(discrim : small_int) is record
   5|          s1 : string(1 .. discrim);
   6|          s2 : string(1 .. 2 * discrim);
                                  ^1
*** 1 Error 2204: A discriminant may not be used in this
***     expression  (3.7.1 (6))
*** Next diagnostic is on line 15
   7|    end record;
   8|  .
   9|    x : rec(5);
  10|begin
  11|    x := (6, "12345", "abcde");
  12|exception
  13|    when constraint_error =>
  14|          null;
  15|    when numeric_error | constraint_error =>
                                ^1
*** Previous diagnostic was on line 6
*** 1 Error 3112: A given exception may only appear once
***     in a handler (11.2 (5))
  16|          null;
  17|end sample_program;
  18|

*** Last diagnostic was on line 15

*** Errors: 2, Warnings: 0
```

The message chain is especially helpful when working with large listings.

Whether on the standard error output or in a listing file, the list of messages is followed by a summary line containing a count of the number of errors in each severity class, for example:

```
? Errors: 2, Warnings: 1
```

3.6.3. Listing Summary

When a listing file is produced, the numbered listing pages are preceded by a summary page that presents data and statistics about the compilation. The following is an example of the summary page.

```
Listing Summary Report

Date of listing:      22-JUL-1989 17:38:14.00

Source file name:     USER01:[ADA.VMS]FOO2.ADA;2
Source file lines:    52
Compiled by:          JONES
Command line used:    ADAC30/LIST=ALWAYS FOO2.ADA
Code size produced:   154 bytes
Data size produced:   0 bytes

Elapsed real time:    00:00:21.80
Elapsed CPU time:     00:00:06.95
Buffered I/O:         253
Direct I/O:           236
Page faults:          1213
```

3.6.4. Message Generation

The Tartan Ada Compiler has many internal phases, any one of which can issue diagnostic messages. Messages are collected in memory until the time comes to generate the message listing. At that time, all the messages are sorted by their position in the source program and are printed.

When you examine a program listing containing many messages, remember that the order in which the messages appear in the listing is not necessarily the order in which the messages were generated. This fact may be important when one error causes another. It is advisable to start correcting your program according to the messages having the lowest numbers and work towards the higher numbers, making an intermediate compilation if necessary.

3.6.5. About Syntax Errors and Recovery

The Tartan Ada Compiler incorporates a parser which is capable of analyzing and correcting all syntactic errors in the source program. This section describes the various error messages that may be issued by the parser. When a syntax error is detected, no object code is generated.

The parser divides the source program text into lexical elements, or *tokens*, such as identifiers, reserved words, constants, etc. When the parser encounters a token that it does not expect, it issues an error message that indicates the position at which the error was detected and the action that was taken to correct the error. Here are some examples of the recovery actions:

In the example below, the trailing "$" does not match any of the valid tokens of Ada and so the parser *deletes it.*

```
1|procedure bad_syntax is
2|    subtype byte is integer range 0 .. 255; $
                                               ^1
***   1 Error 104: Ill-formed token deleted.
3|    x : integer;
```

The compiler also deletes a token occupying an inappropriate place. In the following example, it deletes the superfluous token "while".

```
5|begin
6|    for while i in 1 .. 10 loop
          ^1
***   1 Error 121: Parse error; token deleted.
7|        x := x + 1;
```

In the following example, the missing symbol ";" is inserted, and parsing continues undisturbed.

```
4|    i : integer
5|    b : boolean;
      ^1
***  1 Error 120: Parse error; token ";" inserted.
6|begin
```

In the following example, the syntactically incorrect symbol ":" is replaced by the proper symbol ";".

```
17|    end loop;
18|    x := 15:
              ^1
***  1 Error 122: Parse error; this token deleted, ";" inserted.
19|
```

3.6.5.1. Multi-Token Insertion

The parser may also insert several tokens in an attempt to repair the constructs whose closing tokens (e.g., "end if;") are missing. An example of this recovery is:

```
10|         end if;
11|     end loop;
            ^1,2,3
***  1 Error 120: Parse error; token "if" inserted.
***  2 Error 120: Parse error; token ";" inserted.
***  3 Error 120: Parse error; token "end" inserted.
12|end bad_proc;
```

In this example an additional "end if;" was missing. Note, however, that the maximum number of tokens that can be inserted in succession is limited.

3.6.5.2. Complex Recovery Strategy

If all the simple fix-up and multi-token insertion techniques above are unsuccessful, the parser attempts a more massive correction by deleting many successive or preceding phrases of the program. Foe example,

```
5|    y := 6;    case bad_case is
                  ****************
6*                where x=y => x := 3;
                  ^1
***  1 Error 123: Parse error; ill-formed "<statement>".
7*                when 2 := x = 5;
```

The caret (^) locates the place where the parser detects an error. The compiler indicates the elided portion of the source program by underlining with asterisks. In the first (and possibly the last line), only the tokens deleted are underlined. The lines following the first line are not underlined, but when the entire line is deleted, the symbol after the line number (e.g., lines 6 and 7) changes from a vertical bar (|) to an asterisk. (The above case statement contained three errors. The closing "end case;" was also deleted.)

In cases like the above example, when the parser has deleted more than one token, the error message is

```
Error 123: Parse error; ill-formed "<name>".
```

The name contained within the angle bracket pair is that of the grammatical element that the parser expected to find in this position. Occasionally, the deletion of tokens starts at a point textually preceding the reported error because no legal interpretation of an enclosing construct can be found due to the error within the construct.

Under rare circumstances, you may see an error message

```
Error 127: Parse error; unexpected end-of-file.
```

pointing to a token within the program, with the rest of the program marked as deleted. This message points to the position in the program where the parser detected a syntax error. It indicates that, despite all attempts, the error recovery was unsuccessful until the end-of-file was reached. In this case, correct all the errors reported and examine the program for missing keywords that end complicated syntactic constructs; also, especially examine the few lines that precede and follow this message for syntactic errors.

Chapter 4

The Ada Program Library

The Tartan Ada VMS C30 Program Librarian (ALBC30) implements the Ada Language requirement for separate compilation and dependency control. The program library directory holds all necessary compilation units, including packages that are part of the application under development and any standard packages such as those for I/O.

The *library administration file* is a single file ADA.DB that records the dependencies among these units and their compilation history.

The term *Ada program librarian* refers to executable code that manipulates the library; that is, subcommands of the library command ALBC30 that are discussed in this chapter.

A compilation unit in a library (library unit or secondary unit) is identified by its *Ada-name*, which is either a simple name (an identifier) or the simple name of a subunit and the name of its ancestor. More information about Ada compilation units and program libraries is given in Chapter 10 of the Ada Language Reference Manual. The library administration file does not contain the text of compilation units; it contains only references to files that contain the compilation units in their source and compiled forms.

4.1. *THE* ALBC30 *COMMAND*

The ALBC30 command invokes the Ada Program Librarian to perform the following operations:

- Create an Ada program library
- Delete unit(s) from an Ada library or delete the entire library
- Check the closure of a library unit
- Describe the status of a library unit by generating a dependency graph
- Insert a non-Ada object into the library as the body of a package.
- Link an executable image.

The format of the ALBC30 command is

$ ALBC30 *subcommand*[/*qualifier*...] [*parameter*...] [/*qualifier*...]

Each operation is requested through a subcommand. All ALBC30 subcommands except CREATE assume that the user's Ada library exists in the current directory. The following sections discuss the subcommands and their appropriate qualifiers and parameters.

4.2. *THE* CREATE *SUBCOMMAND*

The CREATE command creates an initialized Ada library database file, ADA.DB, and places it in a directory that has been created to hold the library database file and files required by the library, i.e., separate compilation and all compiler-generated files. Standard system and Ada I/O packages are placed in the library directory and references to them are recorded in ADA.DB.

The format of the CREATE command is

$ ALBC30 CREATE [*directory-spec*]

The parameter *directory-spec* specifies the name of the existing subdirectory in which the library is to be located and must be supplied only if the library directory is not the current directory. If the directory specification supplied is not an existing directory, a message will be issued indicating that the directory was not found and the library files will not be created.

The directory specification must contain the name of or a logical name for an existing directory. A device name is optional. If a subdirectory is used, the names of the directory levels should be separated with periods. No wildcard characters are allowed in the directory specification.

4.3. *THE* DELETE *SUBCOMMAND*

The DELETE command deletes the specified Ada entity, depending upon the supplied parameter. The operations that may be performed are:

- Remove the specification, body and all subunits of a library unit from the library
- Remove the body of a library unit and all its subunits from the library
- Remove a subunit from the library
- Remove an entire library

The format of the DELETE command is

    $ ALBC30 DELETE [/*qualifier*...] [*Ada-name*...] [*directory-spec*]

One or more *Ada-name*(s) or a *directory-spec* may be supplied as parameter(s). A qualifier may be supplied to further specify the unit(s) to be deleted from the library or to request an additional action to be taken during the deletion process.

The wildcard characters of "*" and "%" are recognized in an *Ada-name*. The character "*" matches any sequence (including the empty sequence) of characters in an *Ada-name*. The character "%" matches any single character in an *Ada-name*. Wildcards are used to build regular expressions. When one of the qualifiers /BODY, /SPEC, or /SUBUNIT is used, and a wildcard is present in the parameter supplied, every compilation unit in the library whose name matches the regular expression and is included according to the qualifier will be deleted. Wildcards are not allowed with the /LIBRARY qualifier.

The following command qualifiers may be used:

/BODY   Deletes the body and all subunits that have this body as an ancestor. The *Ada-name* supplied as a parameter must be a library unit name. All compiler-generated files are deleted with the unit. Compiler-generated files do not include those created by the librarian with the LINK subcommand. /BODY is the default if no qualifier is supplied. Format:

                    $ ALBC30 DELETE/BODY *library-unit-name*

/LIBRARY   Deletes the entire library identified by a directory name supplied as a parameter, including all compiler-generated files and the file ADA.DB. No wildcards may be used in the directory name. Format:

                    $ ALBC30 DELETE/LIBRARY *directory-spec*

                If the library to be deleted is in the current working directory, the command may be given as:

                    $ ALBC30 DELETE/LIBRARY []

/LOG   Causes a message to be written to the standard output device when a unit is deleted. The /LOG qualifier may be used in combination with any other qualifier.

/SPEC   Deletes the specification, body and all subunits that have this specification as an ancestor from the library. The *Ada-name* supplied as a parameter must be a library unit name. All compiler-generated files are deleted with the unit. Format:

                    $ ALBC30 DELETE/SPEC *library-unit-name*

/SUBUNIT   Deletes the subunit named and all of its subunits, in a transitive manner. Two *Ada-names* must be supplied as parameters: the ancestor and the simple name of the subunit to be deleted [LRM 10.2]. Format:

                    $ ALBC30 DELETE/SUBUNIT *ancestor simple-name*

Example:

To delete:

```
separate (ONE.TWO.THREE.FOUR)    -- unit ONE is ancestor
procedure FIVE is
begin
         null;
end FIVE;
```

from the library, the command is

```
ALBC30 DELETE/SUBUNIT ONE FIVE
```

Example:

The following example illustrates the use of /DELETE with several qualifiers and wildcards where appropriate.

For a library that contains the following units:

```
Package Spec Radar_Screen
Package Body Radar_Screen
Package Spec Radar_Detector
Package Body Radar_Detector
Subprogram Spec A
Separate Subprogram body Radar_Screen.RangeRings
Separate Subprogram body Radar_Screen.Object
Separate Subprogram body Radar_Screen.Object.Size
```

```
ALBC30 DELETE/SPEC *
```
deletes all units in the library, including standard packages, Tartan runtime packages, support packages, and the file ADA.DB. Thus, when used with DELETE, /SPEC * is a synonym for /LIBRARY

```
ALBC30 DELETE/SPEC %
```
deletes subprogram Spec A

```
ALBC30 DELETE/SUBUNIT RADAR_SCREEN *
```
deletes the last three units listed

```
ALBC30 DELETE/BODY RADAR_SCREEN
```
deletes package body radar_screen as well as the last three units listed

```
ALBC30 DELETE/SUBUNIT * *
```
deletes the last three units listed

```
ALBC30 DELETE/SPEC RADAR*
```
deletes all units except Subprogram Spec A

4.4. *THE* DESCRIBE *SUBCOMMAND*

The DESCRIBE command describes the status of a unit by displaying its dependency graph. The information shown includes the name, type, time of compilation, state of residence (local or foreign), source and object file specifications, dependencies (if any), and subunits (if any).

The format of the DESCRIBE command is

```
$ ALBC30 DESCRIBE[/qualifier...]  [Ada-name...]
```

where one or more *Ada-name*(s) supplied as parameters specify the library unit or subunit to be described. A qualifier may be supplied to further specify the unit(s) to be described or to request an additional action to be taken as part of the process.

The wildcard characters of "*" and "%" are recognized in an *Ada-name*. The character "*" matches any sequence (including the empty sequence) of characters in an *Ada-name*. The character "%" matches any single character in an *Ada-name*. Wildcards are used to build regular expressions. When one of the qualifiers /BODY,

/SPEC, /ANY or /SUBUNIT is used, and a wildcard is present in the parameter supplied, every compilation unit of the specified category in the library whose name matches the regular expression will be described. Additionally, the qualifier /ALL can be used with no parameters.

The following command qualifiers may be used:

/ANY   Describes both the specification and body units for the specified library unit, but will not describe subunits. The *Ada-name* supplied as a parameter must be a library unit name. /ANY is the default if no qualifier is supplied. Format:

$ ALBC30   DESCRIBE/ANY   *library-unit-name*

/ALL   Describes all library units and subunits. No Ada-name should be specified. Format:

$ ALBC30   DESCRIBE/ALL

/BODY   Describes only the body unit for the specified library unit. The *Ada-name* supplied as a parameter must be a library unit name. Do not use in conjunction with /SPEC; to get descriptions of both body and specification, use /ANY. No information about bodies resulting from generic instantiations is given when the /BODY qualifier is given. To get such information use /SPEC or /ANY. Format:

$ ALBC30   DESCRIBE/BODY   *library-unit-name*

/OUTPUT[=*file-spec*]
                 Places the output of this command in the file specified. If no file has been given, output defaults to a file having the name DEPENDENCY.TXT. The default output device is the user's terminal or log file (if invoked from a batch job).

/SPEC   Describes only the specification unit for the specified library unit. The *Ada-name* supplied as a parameter must be a library unit name. Do not use in conjunction with /BODY; to get descriptions of both body and specification, use /ANY. Format:

$ ALBC30   DESCRIBE/SPEC   *library-unit-name*

/SUBUNIT·   Describes a subunit which must be identified by two parameters, the name of the ancestor and the simple name of the subunit [LRM 10.2]. Format:

$ ALBC30   DESCRIBE/SUBUNIT   *ancestor simple-name*

Example:
To describe:

        separate (ONE.TWO.THREE.FOUR)   -- unit ONE is ancestor
        procedure FIVE is
        begin
                    null;
        end FIVE;

the command is

        ALBC30 DESCRIBE/SUBUNIT ONE FIVE

Examples:

1. The ALBC30 DESCRIBE/SPEC command displays the status of the specification of the library unit adatime.

    $ ALBC30 DESCRIBE/SPEC adatime

displays

```
Name:  adatime
Type:  Package Spec
Time:    2-AUG-1988 15:47:05.00
Residence:  local
Source File Name:  SRC:ADATIME.ADASPEC
Symbol File Name:  adatime.di
Dependencies:  none
```

2. The ALBC30 DESCRIBE/BODY command displays the status of the body of the library unit adatime.

```
$ ALBC30 DESCRIBE/BODY adatime
```

displays

```
Name:  adatime
Type:  Package Body
Time:    2-AUG-1988 15:47:05.00
Residence:  local
Source File Name:  SRC:ADATIME.ADABODY
Cbject File Name:  adatime.TOF
Dependencies:
        Type:  Package Spec
        Name:  adatime
Subunits:  none
```

3. The following example illustrates the use of /DESCRIBE with wildcards in a library containing subunits. For a library that contains the following units:

```
Package Spec Radar_Screen
Package Body Radar_Screen
Package Spec Radar_Detector
Package Body Radar_Detector
Subprogram Spec A
Separate Subprogram body Radar_Screen.RangeRings
Separate Subprogram body Radar_Screen.Object
Separate Subprogram body Radar_Screen.Object.Size
```

```
ALBC30 DESCRIBE/ALL
                Will describe all of the compilation units in the library

ALBC30 DESCRIBE/ANY *
                Will describe the first five units

ALBC30 DESCRIBE/SUBUNIT * *
                Will describe the last three units

ALBC30 DESCRIBE/ANY RADAR*
                Will describe the first four units

ALBC30 DESCRIBE/SUBUNIT RADAR* SI*
                Will describe Radar_Screen.Object.Size

ALBC30 DESCRIBE/ANY %
                Will describe Subprogram Spec A

ALBC30 DESCRIBE/SUBUNIT RADAR_DETECTOR *SIZE
                Will not describe any unit
```

4.5. CLOSURE

Closure is defined as the total set of compilation units upon which a unit is dependent. Dependency is conveyed by means of the context (with) clause: a library unit in a context clause of a compilation unit must be compiled before the compilation unit. Additionally, a library unit must be compiled before its corresponding body (secondary unit), and a parent unit must be compiled before any of its subunits. These dependencies are transitive; that is, a library unit in the context clause of a library unit in a context clause of a compilation unit is in the compilation unit's closure, and so on. In other words, the collection of units that must be compiled before a compilation unit is the closure of that compilation unit.

Another cause for dependencies arises from generics. A compilation unit that depends on a generic unit or a unit that contains generics is dependent on the body of that unit, if instantiations of the generic constructs occur. While normally only library and parent units can be in the closure of a compilation unit, this special situation can cause other secondary units to become part of the closure of a compilation unit (See also Section 5.9.2).

The closure of a compilation unit is *consistent* if all units in the closure of the compilation unit are in the library, i.e., they have indeed been compiled in an order that does not violate the partial ordering determined by the dependency relationships. Compilation units may be compiled in any order that is consistent with this partial ordering.

The fundamental rule to be remembered is: A unit can be compiled only if its closure is consistent.

Examples:

1.

```
-- Specification for package MODE3
package MODE3 is

...
end MODE3;

-- Specification for package FOE
with MODE3;
package FOE is

...
end FOE;

-- Specification for package MODE3
with FOE;
package MODE3 is

...
end MODE3;
```

The resubmission of package MODE3 is rejected. A circularity exists between the two package specifications that violates the dependency rules given above:

- MODE3 has FOE in its context clause, which means FOE must be compiled before MODE3.

- FOE, in turn, has MODE3 in its context clause, which means MODE3 must be compiled before FOE.

In other words, MODE3 would have to be compiled before MODE3—an impossiblity.

2. If the following units are submitted to the compiler in the order shown, the closure of package body CROSS_SECTION is consistent.

```
-- Specification for package FOE
package FOE is
...
end FOE;

-- Specification for package FRIEND
package FRIEND is
...
end FRIEND;

-- Specification for package CROSS_SECTION
package CROSS_SECTION is
...
end CROSS_SECTION;

-- Body for package CROSS_SECTION
with FRIEND;
with FOE;
-- NOTE:   The order expressed in the context clause is irrelevant
--   to the compilation order.
package body
    CROSS_SECTION is
...
end CROSS_SECTION;
```

At this point, a recompilation of package FOE would cause the closure of package body CROSS_SECTION to become inconsistent.

The *full closure* of a unit is defined as the set of units in the unit's closure, their corresponding bodies and all of their subunits, and the full closure of each of these units. Establishing the full closure is recursive because Ada allows secondary units to have dependencies that are in addition to the dependencies of the corresponding library unit or parent unit. Thus a secondary unit may bring additional library units and their bodies into the full closure.

The full closure of a compilation unit is said to be *consistent* if all the units in the full closure of the compilation unit are in the library and have been compiled in an order that does not violate the partial ordering determined by their respective closures.

The fundamental rule to be remembered is: A unit constituting a main program can be linked only if it has been successfully compiled and its full closure is consistent. For a successful compilation of the unit only a consistent closure, but not a consistent full closure is required. ·

Example:

The full closure of the unit CROSS_SECTION is consistent because all required compilation units have been compiled into the library in a correct order.

```
    -- Specification for package FRIEND
    package FRIEND is
    ...
    end FRIEND;

    -- Body for package FRIEND
    package body
        FRIEND is
    ...
    end FRIEND;

    -- Specification for package FOE
    package FOE is
    ...
    end FOE;

    -- Body for package FOE
    package body
        FOE is
    ...
    end FOE;

    -- Specification for package RADAR_DETECTOR
    package RADAR_DETECTOR is
    ...
    end RADAR_DETECTOR;

    -- Body for package RADAR_DETECTOR
    package body
        RADAR_DETECTOR is
    ...
    end RADAR_DETECTOR;

    -- Specification for package CROSS_SECTION
    package CROSS_SECTION is
    ...
    end CROSS_SECTION;

    -- Body for package CROSS_SECTION
    with FRIEND;
    with FOE;
    package body
        CROSS_SECTION is
    ...
    end CROSS_SECTION;
```

4.6. *THE* CLOSURE *SUBCOMMAND*

The CLOSURE command checks the closure or full closure of a unit by displaying the partial ordering of its closure.

The format of the CLOSURE command is

   $ ALBC30 CLOSURE [*/qualifier...*]   [*Ada-name...*]

where one or more *Ada-name*(s) supplied as parameters specify the unit or subunit whose closure is to be checked. A qualifier may be supplied to further specify the unit(s) to be described or to request an additional action to be taken as part of the process.

The wildcard characters of "*" and "%" are recognized in an *Ada-name*. The character "*" matches any sequence (including the empty sequence) of characters in an *Ada-name*. The character "%" matches any single character in an *Ada-name*. Wildcards are used to build regular expressions. When one of the qualifiers /BODY,

/SPEC, /FULL or /SUBUNIT is used, and a wildcard is present in the parameter supplied, the closure or full closure of every compilation unit of the specified category in the library whose name matches the regular expression will be checked.

The following command qualifiers may be used:

/BODY   Checks closure only on the body unit named by the *Ada-name* parameter. The *Ada-name* supplied as a parameter must be a library unit name. Format:

> $ ALBC30   CLOSURE/BODY   *library-unit-name*

/FULL   Checks the full closure (linking closure) for the specification unit named by the *Ada-name* parameter. The *Ada-name* must be a library unit name. Format:

> $ ALBC30   CLOSURE/FULL   *library-unit-name*

/OUTPUT[=*file-spec*]

> Places the output of this command in the file specified. If no file has been given, output defaults to a file having the name CLOSURE.TXT. The default output device is the user's terminal or log file (if invoked from a batch job).

/SPEC   Checks closure only on the specification unit named by the *Ada-name* parameter. The *Ada-name* must be a library unit name. /SPEC is the default if no qualifier is supplied. Format:

> $ ALBC30   CLOSURE/SPEC   *library-unit-name*

/SUBUNIT   Checks closure of a subunit which must be identified by two parameters, the name of the ancestor and the simple name of the subunit [LRM 10.2]. Format:

> $ ALBC30   CLOSURE/SUBUNIT   *ancestor simple-name*

Example:
To check closure for:

```
separate (ONE.TWO.THREE.FOUR)    -- unit ONE is ancestor
procedure FIVE is
begin
         null;
end FIVE;
```

the command is

> ALBC30 CLOSURE/SUBUNIT ONE FIVE

Examples:

1. The ALBC30 CLOSURE/SPEC command displays the status of the specification of the library unit DISPLAY_TARGETS.

> $ ALBC30 CLOSURE/SPEC display_targets

displays

```
     Closure partial ordering at 16-JUL-1987 10:28:32.00 for
        implicit Subprogram Spec   display_targets


     Error   Type                           Unit
     ------------------------------------------------------------------

              Package Spec                  paintscreen
              Package Spec                  cross_section
              Package Spec                  radar_detector
              implicit Subprogram Spec      display_targets


     Info  6011: The files required for compilation by implicit
        Subprogram Spec display_targets are consistent (10.3).
```

2. The ALBC30 CLOSURE/FULL command displays the partial ordering of the full closure of the body of the library unit DISPLAY_TARGETS in a situation where the package body for PAINTSCREEN is obsolete.

```
     $ ALBC30 CLOSURE/FULL display_targets
```

displays

```
     Full Closure partial ordering at 16-JUL-1987 10:28:32.00 for
        implicit Subprogram Spec   display_targets

     Error  6005: recompilation of Package Spec paintscreen has made
        dependent Package Body paintscreen obsolete (10.3 (5)).


     Error   Type                           Unit
     ------------------------------------------------------------------

           ·  Package Spec                  paintscreen
              Package Spec                  cross_section
              Package Spec                  radar_detector
        *     implicit Subprogram Spec      display_targets
              Package Body                  paintscreen
              Package Body                  cross_section
              Package Body                  radar_detector
              Subprogram Body               display_targets


     Info  6009: The files required for linking by implicit Subprogram
        Spec display_targets are not consistent (10.3).
```

4.7. THE FOREIGN_BODY SUBCOMMAND

A subprogram written in another language can be called from an Ada program. Pragma FOREIGN_BODY is used to indicate that the body for a library unit that is a non-generic package specification is provided in the form of an object module. This pragma allows subprograms within that package to be called from an Ada program. See Section 5.1.2.2 for a description of the use of and restrictions on pragma FOREIGN_BODY.

Prior to successful linkage, the body of the respective package must be provided to the library. The FOREIGN_BODY command allows the programmer to introduce a reference to the compiled body of a package (in an object file) into the library. The format of the FOREIGN_BODY command is

```
     ALBC30 FOREIGN_BODY Ada-name file-spec
```

where the parameter *Ada-name* specifies the unit within the library for which a body is being provided. The parameter *file-spec* specifies the name of compiled package body or object library of subprogram bodies to be

used. A reference to this file specification (by means of the supplied Ada-name) will be entered into the current Ada library. The file specification must contain the file name and type. No wildcard characters are allowed in the file specification.

If a specification for a foreign body is deleted from the program library, the database entry for the body is also deleted. The object file is *not* deleted.

Example. The ALBC30 FOREIGN_BODY command replaces the standard package adatime with a reference to a timing module written in FORTRAN.

```
$ ALBC30 FOREIGN_BODY adatime USER01:[PROJECT]fortran_time.TOF
```

A pointer for this package in the file USER01:[PROJECT]fortran_time.TOF has now been inserted in the ADA.DB file.

4.8. *THE* LINK *SUBCOMMAND*

The LINK command checks that the unit within the library specified by the user has the legal form for a main unit, checks all its dependencies, finds all required object files, and links the main program with its full closure (See Section 4.5) producing an executable image. The format of the LINK command is

```
$ ALBC30 LINK[/qualifier...] library-unit-name [/qualifier...]
```

where the parameter *library-unit-name* specifies the unit in the library to be made the main program and must be supplied.

The ALBC30 LINK command calls the Tartan Linker which is documented in *Object File Utilities for the Tartan Ada VMS C30*, Chapter 2. The output file from the Tartan linker is *library-unit-name*.XTOF.

The following qualifiers may be used with the ALBC30 LINK command:

/ALLOCATIONS   Produce a link map showing the section allocations. This qualifier may be used only if the Tartan Tool Set has been selected when creating the library (See Section 4.2).

/CONTROL=*file-spec*
    Specifies a file used to pass instructions and qualifiers to the Tool Set Assembler and Linker programs. This file may be used to specify what Toolset components, commands or versions of components are to be used in building the final program. No wildcard characters are allowed in the file specification. See *Tartan Ada Object File Utilities Manual*, for information in linker control files.

/EXECUTABLE [=*file-spec*] Controls the name of the executable image created by the Tool Set linker. The default file name is that of the main program. No wildcard characters are allowed in the file specification.

/KEEP   Retain the elaboration order and link control files after the linking operation is complete. The program may then be relinked using The Tartan Linker, TLINK, independent of the librarian (See *The Tartan Object File Utilities Manual*, Chapter 2). This method is primarily a debugging tool. The user assumes full responsibility for the consistency of the program when it is used instead of using the ALBC30 LINK command.

/MAP   Produce a link map containing all information except the unused section listings.

/RESOLVEMODULES
    Include all sections from the specified modules in the link. This action corresponds to the "usual" behavior of other linkers. Some linker analysis is eliminated at the expense of a (possibly) larger output module.

/SYMBOLS   Produce a link map showing global and external symbols.

/UNUSED   Produce a link map showing the unused sections.